HighSpeed Pascal

HiSoft
High Quality Software

User Manual

Published by HiSoft

The Old School, Greenfield, Bedford MK45 5DE UK

First Edition, January 1992 - ISBN 0 948517 50 6

Preface to HighSpeed Pascal

Introduction

HighSpeed Pascal is a complete package for the production of fast, efficient Pascal programs on your Amiga® computer.

To help you use the package easily and efficiently, we have split the documentation for HighSpeed Pascal into two volumes; this is the User Manual which covers the use of the package on your computer and has chapters detailing the editor, compiler and debugger with a general overview of the language. The accompanying Technical Reference consists of a language reference chapter, documentation for the supplied units, details of the error messages produced and technical information on the internal workings of HighSpeed Pascal.

The manuals aim to cover all aspects of installing and using HighSpeed Pascal on your Amiga® - they do not attempt to teach you 680x0 programming or the Pascal language although the examples should be of assistance in this regard. For further reading, you should consult the *Bibliography*. You may wish to take advantage of our special offers on a range of technical books that you will find invaluable if you are a serious programmer; for details, see the order form included with this package.

Please spend some time and effort getting to know and learning how to use the manual so that you can gain the maximum benefit from HighSpeed Pascal.

The rest of this section explains how to use the manuals, whether you are a beginner or an expert, how to use your computer to best effect with HighSpeed Pascal and, finally, we outline the different typestyles that we have used throughout the manuals to (hopefully) make them easy and enjoyable to use.

User Manual

HighSpeed Pascal for the Amiga®

By HiSoft and D-House

© Copyright 1991 HiSoft and D-House. All rights reserved.

Program:

designed and programmed by HiSoft, Christen Fihl and D-House

Manual:

written by David Nutkins, Alex Kiernan and Keith Wilson

This guide and the HighSpeed Pascal program diskettes contain proprietary information which is protected by copyright. No part of the software or the documentation may be reproduced, transcribed, stored in a retrieval system, translated into any language or transmitted in any form without express prior written consent of the publisher and copyright holder(s).

HiSoft shall not be liable for errors contained in the software or the documentation or for incidental or consequential damages in connection with the furnishing, performance or use of the software or the documentation.

HiSoft reserves the right to revise the software and/or the documentation from time to time and to make changes in the content thereof without the obligation to notify any person of such changes.

Table of Contents

Preface to HighSpeed Pascal 1

Introduction 1

How to use the Manuals 2

A Course for the Beginner 2

A Course for Seasoned Programmers 3

System Requirements 4

Typography 4

Typefaces 4

Acknowledgements 5

Chapter 1 Introduction 7

Disk Contents 7

Program Disk 8

Units disk 9

Always make a back-up 10

Installation 10

Hard disk users 10

Floppy disk users 11

Installation details 11

Registration Card 14

The ReadMe File 14

Chapter 2 A Quick Tutorial 15

Getting Started 15

Loading HighSpeed Pascal 15

Using the menu 16

Creating your first program 18

Compiler errors 19

Using the supplied units 20

The menus 22

Chapter 3 The Editor 25

Introduction 25

The Editor 25

Starting the Editor 27

Requesters and gadgets 27

Menus and sub-menus 34

The Editor's windows 35

Entering text and moving the cursor 39

Bookmarks 42

Deleting text 43

Block Commands 44

Searching 47

Disk Operations 50

Macros 53

Settings 55

Miscellaneous Commands 60

Keyboard command summary 63

Chapter 4 The Compiler 67

Compiling from the editor 67

Overview 68

Program menu 68

Compiler options 73

Pascal Config files 77

Compiling from the command line 79

$ Switch Option 80

Directory Options 80

Mode Options 80

Compiler directives 81

Switch directives 81

Other single letter directives 83

Source code control 84

Chapter 5 The Debugger 85

Introduction 85

Debugging Pascal Programs 86

MonAm Concepts 89

Front Panel Display 89

MonAm and Multi-Tasking 92

Symbolic Debugging 92

MonAm Requesters 93

Command Input 94

MonAm Overview 95

MonAm Reference 98
Numeric Expressions 98
Window Types 103
Cursor Keys 107
Window Commands 108
Other A Commands 113
Screen Switching 114
Breakpoints 115
History 118
Quitting MonAm 119
Loading & Saving 120
Executing Programs 121
Searching Memory 122
Miscellaneous 124
Command Summary 129
Bug Hunting 131
Exceptions 132
Restrictions 134

Chapter 6 Other Tools 137

Libmaker 137
Examples 138
FD to Pascal converter 138
Usage 139
Unit format 139
Using other libraries 140

Chapter 7 Operating System Support 141

The Amiga Units **141**

Conventions 142

Libraries **145**

AmigaDOS 146

DiskFont 148

Exec 148

Expansion 150

Graphics 150

Icon 152

IFFParse 153

Intuition 154

Layers 156

Translator 157

Devices **158**

AmigaPrinter 160

Audio 161

Clipboard 162

Console 163

Gameport 163

Input 164

Keyboard 164

Narrator 165

Parallel 165

SCSI 166

Serial 166

Timer 167

Trackdisk 168

Resources 168
CIA 169
Disk 169
FileSystem 170
Misc 170
Potgo 171
Other 171
Amiga 171
Hardware 172
Keymap 173
Workbench 173
C for Pascal programmers 174
Data structures 174
Programs 182
Appendix A Compatibility 191
HighSpeed Pascal vs Turbo Pascal PC 191
HighSpeed Pascal Amiga vs HighSpeed Pascal Atari 193
HighSpeed Pascal vs ISO Pascal 194
Extensions to ANS Pascal 194
Appendix B Technical Support 197
Appendix C Bibliography 199
Amiga 199
Pascal 200
Algorithms & Data Structures 201

How to use the Manuals

We have designed these manuals to tell you about using HighSpeed Pascal on the Amiga® computers. We have packed a great deal of information about the package into the manuals and, in order to help you use them efficiently and easily, we will now plot recommended courses through the manual for you, whether you are a beginner to Pascal or a seasoned expert.

A Course for the Beginner

If you are a newcomer to the Pascal language then we recommend that you read one of the books in the *Bibliography* alongside the manuals.

Chapter 1 is an introduction to using HighSpeed Pascal and covers the contents of your master disks, making a back-up copy of them, installing the package and registering your purchase - you *must* read this first.

Following this introduction, there is a simple tutorial which you should then follow to familiarise yourself with the use of the main parts of the program suite.

Chapter 3 considers the editing environment with an overview of using the package and is well worth reading; much of *Chapter 4*, detailing the compiler, is liable to mean little until you become more experienced but should be used as a reference. The overview of the debugger in *Chapter 5* is recommended, though the detail of this package can be left for a while.

Chapter 6, on the various other tools supplied with HighSpeed Pascal, can be left until you need it.

You will need to read *Chapter 7*, which discusses the Amiga-specific units, when you come to start programming in earnest.

The *Technical Reference* manual should be used to refer to specific technical aspects of the Pascal language and of the general units supplied - this will be useful when you have become familiar with using the system. *Chapter 1* (which discusses the implementation of the Pascal language in this package) will be useful in conjunction with whichever tutorial book you choose to read.

The *Appendices* are mainly for reference and you will only need to dip into them occasionally.

We hope you find HighSpeed Pascal easy and friendly to use, please do not hesitate to write to us with any suggestions for improvements and/or alterations.

A Course for Seasoned Programmers

If you are experienced in the use of the Pascal language and the Amiga, you will want to use both the manuals as a reference to HighSpeed Pascal and its implementation on the Amiga computer.

However, we recommend that you read the first three chapters (this *Preface*, the *Introduction* and the *Tutorial*) before rushing into programming.

Unless you already have HiSoft Devpac 3, it is also worth having a good look at *Chapter 3* which details the editor and the integrated environment since this is where you are likely to spend most of your time.

You can then dip into *Chapter 4* (the *Compiler*), *Chapter 5* (the *Debugger*), *Chapter 6* (*Other Tools*) and *Chapter 7* (the Amiga-specific units) as and when you need them.

Use the *Technical Reference* manual to find out particular differences between HighSpeed Pascal and other systems that you may have used previously.

If you have any problems *please* read the relevant section of the manual before contacting us for technical support.

The *Appendices* are for general reference and it is worth glancing through all of them to acquaint yourself with their contents.

Good luck, we hope you find HighSpeed Pascal a powerful, flexible and easy-to-use development system. Of course, we welcome any written comments you may have on how we might improve both the program and the manuals.

System Requirements

HighSpeed Pascal will run on any Amiga® 680x0 computer (A500, A1000, A2000, A3000 etc.) with at least 512Kb of memory and a disk drive. You will undoubtedly find it useful for this and other programs to purchase extra memory, a second disk drive and/or a hard disk.

Users with only 512Kb of RAM may run out of memory when attempting to compile larger programs or in other circumstances. Some advice for users with only 512Kb memory is given in the next chapter. Upgrades to a megabyte of memory are available at very reasonable prices and we strongly recommend this, not just for HighSpeed Pascal but for general use too.

Typography

In order to make the manual easy to read and to convey the maximum information as clearly as possible, we have adopted certain typefaces and typestyles throughout the manual.

Typefaces

Palatino	General text.
Futura oblique	Chapter and sub-Chapter headings and references to them.
Monospace	Used to show something that is typed in at the keyboard or displayed on the screen. Predominantly used in program listings and references to function names, variables etc.
Avant Garde	Used for filenames, menu selections and button names. Also used to denote legends on single keys such as Alt (the Alternate key) and Ctrl (Control).

Typestyles

The *Italic* style is used mainly for *emphasis*.

Special Characters

[ ]	Within syntax descriptions, information enclosed in [ ] is optional.
...	Indicates repetition in syntax descriptions.
. . .	Vertically-spaced dots show that some part of a program has been omitted.

Acknowledgements

The trademarks (both registered and otherwise) of various companies are used throughout this manual. In particular:

HighSpeed Pascal, Power BASIC, HiSoft BASIC, GenAm and *MonAm* are trademarks of HiSoft.

Amiga® is a registered trademark of Commodore Inc. *AmigaDOS, Kickstart, Intuition, Workbench* are trademarks of Commodore Inc.

Turbo Pascal® is a registered trademark of Borland Inc.

We acknowledge any other trademark used but not listed above.

We would like to thank the following people for their invaluable help in the production of HighSpeed Pascal and the manuals:

Christen Fihl for his hard work in putting together the compiler, Martin and Jacob from D-House for setting us a good example, all our ß-testers for their patience and Julia, Sallie, Pauline and Marlynne for keeping us smiling through the bad times.

Chapter 1
HighSpeed Pascal Introduction

Disk Contents

HighSpeed Pascal is supplied on three 3.5" disks.

The first two disks are *Program Disks* containing a Workbench, an HSPascal drawer, within which the tools can be found, and many example programs. You may boot your Amiga® from this disk and proceed to use HighSpeed Pascal immediately although you *must* ensure that you use the correct version (1.3 or 2.0) for your machine.

The third disk contains the full set of HighSpeed Pascal 'unit' files including those required for accessing the Amiga® operating system. These can also be found on the *Units Disk* as a Pascal.lib file for compactness and speed. Many users will find this sufficient for their requirements although more advanced users are urged to re-make this file as needed.

As a reference for the units, some Pascal source giving the units' interface section is also supplied on the *Units Disk*.

Please note that the following list of files is intended as a guide only and for the actual disk contents you should refer to the Contents file. Subsequent versions of HighSpeed Pascal may contain extra files.

Program Disk

Each *Program Disk* is a cut-down bootable Workbench disk containing the development tools and example programs. Some directories and files of particular interest are:

Contents — The disk contents list for your version of HighSpeed Pascal.

ReadMe — A text file including the latest details about HighSpeed Pascal. Please read this file carefully before contacting our technical support department with any queries.

HSPascal/ — This drawer contains the program tools detailed below and the examples drawer.

Libs/arp.library — (1.3 version only). An additional library containing the file requester used by the HSPascal editor.

Libs/iffparse.library — Not normally distributed with Workbench 1.3, this library may be used by your own programs to read and write IFF files.

S/Startup-Sequence — The script file run automatically by AmigaDOS as part of the boot process. Advanced users may wish to modify this file in order to customise the disk.

S/User-Startup — (2.0 version only) Script run by the Startup-Sequence where all modifications should go.

Prefs/Env-Archive/HSPascal — The contents of the ENVARC: directory are automatically copied into ENV: at boot time by the Startup-Sequence script file. You may place preferences files or default icons within this directory (see the editor chapter for details).

The HSPascal drawer

This drawer contains the HighSpeed Pascal programs and may be copied onto your hard drive as part of the installation procedure. Each of these programs is documented in detail elsewhere in this manual. Where disk space is limited, some tools may be found on the *Units Disk*.

HSPascal	The multi-window editor and control program.
HSPascal.prefs	Editor preferences.
HSPC	The powerful HighSpeed Pascal compiler.
Pascal.cfg	The default compiler options file.
MonAm	The debugger.
MonAm.libfile	A support file for the debugger enabling the automatic naming and display of many operating system library calls.
Demos/	Many example Pascal programs can be found in the Demos drawer. These are Turbo Pascal compatible programs.
AmigaDemos/	This drawer contains Amiga specific examples that provide a useful introduction to using the operating system with HighSpeed Pascal.
LibMaker	A utility program which allows you to add further units to Pascal.lib or create a new one.
FDtoPascal	A utility program which generates the Pascal unit needed to interface to an extension library from a standard FD file.

Units disk

All of the HighSpeed Pascal unit files can be found on this disk both individually and as part of a Pascal.lib library, along with the source to their interface sections for reference purposes.

Pascal.lib	Pre-compiled library file containing all of the HighSpeed Pascal units. By default, this is read before compiling the first program.
Units/	A directory containing the Turbo Pascal compatible and Amiga operating system units in compiled form.
Interface/	Pascal source to the interface modules of the supplied units. These cannot be compiled, they are for reference only!

Always make a back-up

Before using HighSpeed Pascal you should make a back-up copy of the distribution disks and put the originals away in a safe place. They are not copy-protected to allow easy back-up and to avoid inconvenience. The disks may be backed-up using the Workbench or any back-up utility.

Before hiding away your master disks make a note in the box below of the serial number. You will need to quote this if you require technical support.

Serial No:

Installation

The installation procedure is fairly straightforward. We advise you to read the following section carefully, referring to your HighSpeed Pascal disk contents and ReadMe file for the latest information.

Hard disk users

Our recommended installation procedure for HighSpeed Pascal on a hard drive is to create an HSPascal drawer in the root of your hard drive which contains all the relevant files. This keeps your drive tidy and makes it easy to upgrade to future versions.

The easiest way to do this is from Workbench. After booting into Workbench from your hard drive, perform the following steps:

- Drag the HSPascal drawer icon from the appropriate *Program Disk* onto your hard disk.
- Drag all icons from the *Units Disk* and place them *inside* the newly created HSPascal drawer.
- Run the editor from your hard drive by double-clicking on the HSPascal program icon.
- Choose Settings... from the Settings menu, select the Pascal.lib within the HSPascal drawer on your hard disk and use Save Settings.

- Select Pascal Config... from the Settings menu, update the unit search path to refer to the Units drawer on your hard disk and click on Save.

The HSPascal directory can be added to the AmigaDOS command search path via the Startup-Sequence or User-Startup to allow convenient use from the Shell, CLI or script files.

Workbench 1.3 users may also wish to copy the libs/arp.library file (containing the file requester used by the editor) to the assigned LIBS: directory on their hard disk, if none is already present.

Floppy disk users

Floppy disk users are recommended to use their backup of the appropriate *Program Disk* as a basis for customisation, adding or removing files as necessary. As with most Amiga® development packages, a second floppy disk drive is a considerable advantage, avoiding much disk swapping.

If you have sufficient memory, you may wish to set up the editor so that it automatically loads the compiler, Pascal.lib and debugger into memory at startup so that you can use a separate, formatted disk for your source code.

To convert the Startup-Sequence script file to boot into a Shell window rather than Workbench simply replace the `LoadWB` command with `NewCLI`.

Installation details

ENV usage

ENV: (the environment directory) is used by HighSpeed Pascal for two things; firstly, default settings files may be located there (see the following table) and secondly, default icons are taken from there.

HighSpeed Pascal follows the Workbench 2.0 convention of storing environment variables which survive a reboot in Env-Archive. In order to create default settings or icons, you must not only save them in ENV:HSPascal but also in ENVARC:HSPascal if you wish them to be retained between sessions.

For those unfamiliar with this, ENV: normally resides on the RAM disk and thus is lost each time you reboot. However, in order to set up default icons etc., the startup sequence script files copy the contents of SYS:Prefs/Env-Archive (assigned the name of ENVARC:) to ENV:. Workbench 1.3 users may wish to duplicate this technique in their startup sequences.

Search paths

There are many files which the editor, compiler and debugger search for at various times. The following table serves as a guide to which directories are searched for each file and in which order. You may use this information to move a file from a 'local' directory to a 'global' directory such as copying the MonAm.libfile into LIBS: for system-wide access.

HSPC	(selectable) HSPascal directory
Pascal.lib	(selectable) HSPascal directory
MonAm	(selectable) HSPascal directory
MonAm.libfile	current directory or LIBS:
MonAm.prefs	current directory, ENV:Devpac (or HSPascal directory from the editor)
Pascal.cfg	current directory (ENV:HSPascal or HSPascal directory from the editor)
HSPascal.prefs	current directory, ENV:HSPascal or HSPascal directory
units, include files, object files	selectable via the Pascal Config... requester

In general, the location of tools is selectable (via Settings...) and defaults to the editor directory whilst preferences are searched for in the current (project) directory, the local environment and finally the editor directory. Any amendments to this will be noted in the ReadMe file.

Libraries required

Disk loaded libraries and devices required by the HSPascal editor are listed below. These libraries should be present in the assigned LIBS: directory (or DEVS: for devices), normally upon your boot volume.

asl.library	(2.0 only) file requester
arp.library	(1.3 only) file requester
icon.library	(1.3 only) always required
diskfont.library	always required
clipboard.device	required for cut and paste

Conserving memory

HighSpeed Pascal is intended for use on systems with 1Mb of memory or more. However, it is possible to use the package on a 512Kb Amiga®. In order to do this, or to reduce memory usage when compiling large programs, the following steps can be taken.

- Remove disk buffers and unwanted DOS devices.
- Remove other programs run in the background from the startup sequence.
- Run from CLI instead of Workbench.
- Have the minimum number of windows open at any one time.
- In the editor settings, select loading of tools from disk each time they are used to allow re-use of the memory in between compilations.
- Use a smaller Pascal.lib or none at all.
- Compile to disk rather than memory.
- Split your program into smaller sections or units which are included or referenced by a primary file. When compiling, the bulk of the source need not be in memory.

Further savings can be made by running the editor, compiler and debugger programs individually rather than using the integrated environment. This will also allow you to compile larger programs.

Registration Card

Enclosed with this manual is a registration card which you should fill in and return to us. Without it you will not be entitled to your 30 day free technical support or upgrades within this period. Be sure to fill in all the details especially the serial number and version number.

The ReadMe File

As with all HiSoft products HighSpeed Pascal is continually being improved and the latest details that cannot be included in this manual may be found in the Readme file on the relevant *Program* disk. This file should be read at this point, by double-clicking on its icon from the Workbench. It will also contain last-minute details on the installation process.

Chapter 2
A Quick Tutorial

Getting Started

In this chapter we will try to get you started using the HighSpeed Pascal editor and compiler. You will learn how to load the HighSpeed Pascal system into memory and how to enter, edit, save and run your programs. On top of that there is an introduction to what a *unit* is and how to make your own units.

In this chapter you will be requested to enter four Pascal programs into the editor; none of these programs will be particularly useful, they are merely examples of simple, easy to enter programs.

If you spot any syntactic errors in the listed programs, don't correct them! They were created on purpose, to give you practice developing with the system.

To find out about a particular menu selection or for further detail on compiler options etc., take a look in the editor or compiler chapters.

Loading HighSpeed Pascal

From the supplied disk

HighSpeed Pascal is supplied with two bootable Program disks, one for Workbench 1.3 and the other for Workbench 2.0 machines. Make sure you are using the correct disk for your computer (don't worry, you're not missing out - they both have the same programs on them).

After switching on or resetting your Amiga, insert the correct HighSpeed Pascal Program disk in your disk drive - if you have more than one floppy drive, use the first one. The Amiga Workbench will now start loading.

Once the icons appear, double-click on the Pascal disk icon to open its window, then double-click on the file named HSPascal. This is the HighSpeed Pascal editor.

From a hard drive

If you have a hard drive, you may still want to load from floppy for the purposes of this demo to have a quick peek at the program before installing it on your hard disk.

However, you may have already installed HighSpeed Pascal on your hard disk; if so, double-click on your hard disk icon and then start the HighSpeed Pascal system by double clicking on the file HSPascal. (The recommended method for installation is copying the HighSpeed Pascal file and units into a drawer which you must first find and open).

Running out of memory

If you are unable to follow this demonstration because of insufficient memory, first try closing any other windows which are open both in the editor and on the Workbench. Failing this, it is possible to set up the system so that it compiles to disk instead of memory and loads the compiler each time but for this you will need to refer to the *Editor* chapter.

Using the menu

Look closely at the screen, it consists of two parts: the screen title bar and the editor window. You choose a menu selection by holding down the right mouse button and placing the arrow over a menu title. By doing so, a menu will appear. From this (still holding the mouse button down), point to the choice you want and select it by letting go of the button.

Try this now ...

Keyboard shortcuts

Instead of selecting with the mouse you can use a *keyboard shortcut*. You can see the Amiga key shortcuts to the right of the choices in the pull down menus.

The shortcut for New is A N which means that you should hold down the right hand A key and at the same time press 'N'. This would be the same as selecting New via the mouse.

Other shortcuts are not visible from the menu. For example, the Program menu commands use the Ctrl key for shortcuts. A full keyboard command summary can be found at the end of the *Editor* chapter.

Creating your first program

The HighSpeed Pascal editor starts up with a new and empty window ready for you to type your program. A window with the name `Untitled` will show up on your screen. In the upper left corner you will see the cursor. You are now ready to enter your first HighSpeed Pascal program.

Type these few lines (at the end of every line you press Return to start the next one):

```
Program First;
Begin
  WriteLn('Hello World!');
End.
```

Be sure to check that the first and third lines end with a semi-colon (`;`) and that the last line is ended by a period (`.`). If this is not the case use the mouse or the arrow keys to place the cursor and correct the error(s). Note that whether you use capitals or lower case is unimportant.

Saving to disk

Before you run your program it is often a good idea to save it to disk since if the computer crashes you will lose all of your work. To save and name an untitled file choose Save or Save As... from the Project menu. When you do this a *file requester* will appear on your screen.

Type in the name under which you wish to save your program (e.g. `First.pas`) and press the Return key. Notice the pointer changes to a little stopwatch; this is the 'busy' pointer. When HighSpeed Pascal has finished saving your program you will notice that the title on the editor window now corresponds to the name that you just entered.

Running the program

You are now ready to run your program. Do this by choosing Run from the Program menu (or by pressing the Ctrl key and X at the same time). The editor will first look at the disk to find the compiler then automatically compile and run your program.

A window will appear and a line reading

```
Hello World!
```

will be displayed (if you don't get this then you have probably made a mistake in entering the program - the compiler will tell you where the problem is so that you can correct it and try again).

The computer is now waiting for you to press a key or close the window (under Workbench 2 you can do this with Ctrl-\). Do so and the window will disappear revealing the editor window once again. Now your program has finished executing, you are returned to the place in your file where you left off.

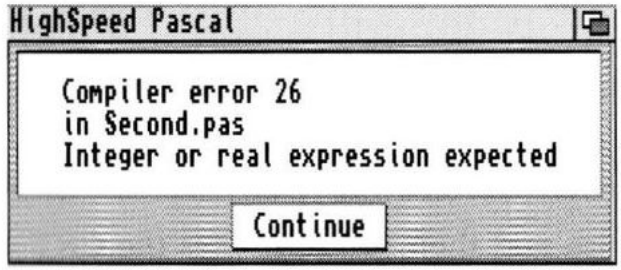
Compiler errors

Open a new editor window by pressing right Amiga and N at the same time (if you are low on memory, close the existing window first). When a new editor window appears start entering this program:

```
Program Second;
Begin
  WriteLn(Abs('a'));
End.
```

Make sure that you entered each line correctly and then name your program by choosing Save or Save As... in the Project menu.

Try to run the program (Ctrl-X); HighSpeed Pascal will immediately stop the compilation and display an error message. Acknowledge the message by pressing the C key to continue. As you can now see from the location of the cursor, the problem is the parameter to the Abs function.



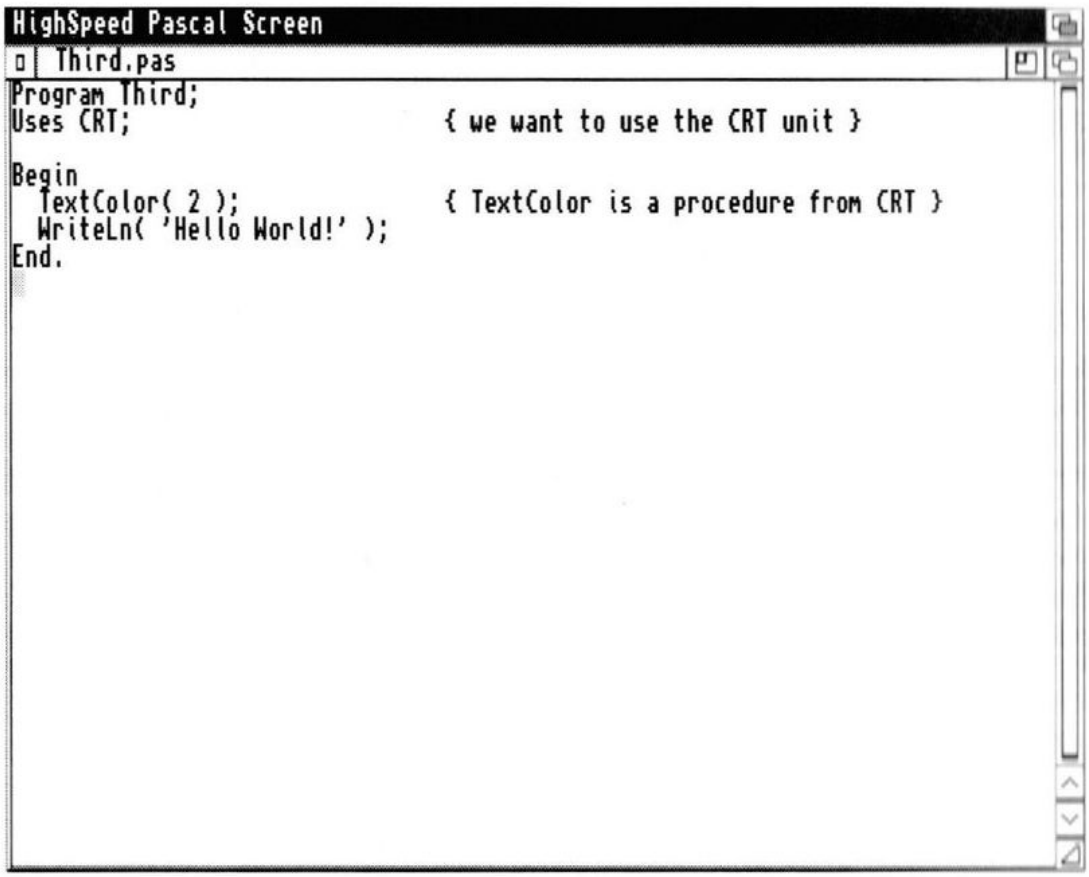
If you check the function documentation, you will notice that the Abs function takes an integer or real type parameter and not a character type one as we used in this program.

Correct the error (by placing the cursor at the end of 'a', hitting Backspace three times and typing -4.56, say); your second program should now run.

Using the supplied units

In this, your third program, you will see how to use units. A unit is a collection of pre-compiled procedures, functions and data. The smart thing about units is that they don't have to be compiled every time you compile a program that uses them; they only have to be linked in with your program. This should encourage you to use units whenever possible, since compilation speed is increased.

Open a new editor window and enter the following program:



Save this program and then run it; it uses the CRT unit.

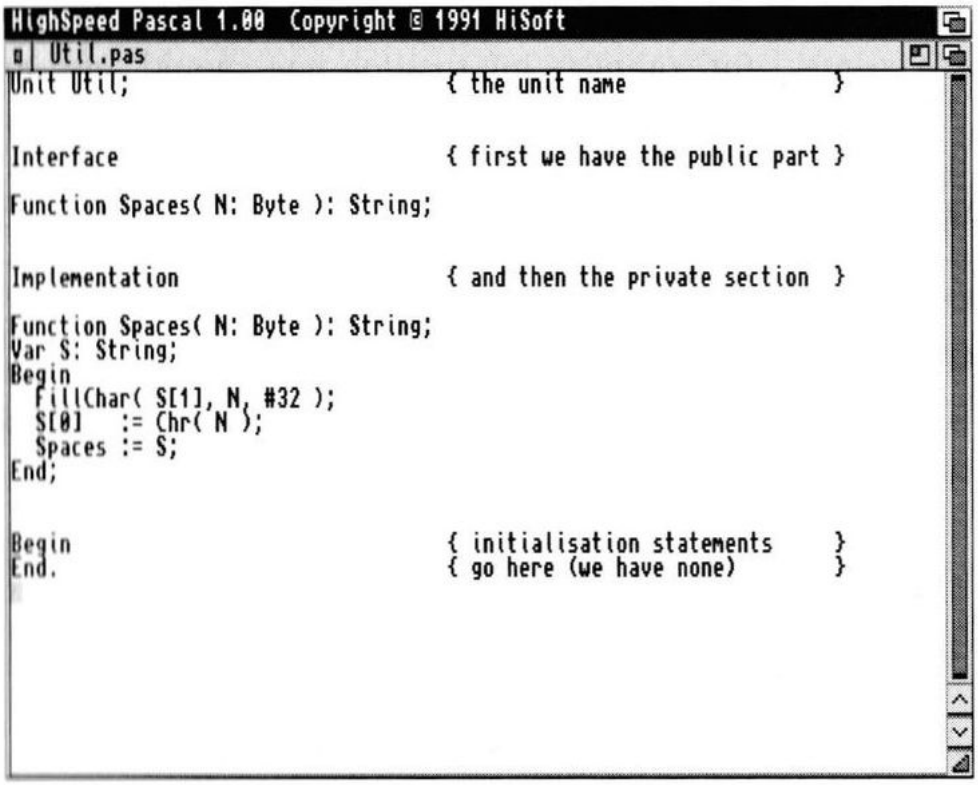
Creating your own unit

Here are the steps necessary to create your own unit. Let's say that you have created a function called Spaces:

```
Function Spaces(N: Byte): String;
Var S: String;
Begin
   FillChar(S[1], N, #32);
   S[0] := Chr(N);
   Spaces := S;
End;
```

You use this function in several of your programs and consequently you have a number of identical copies of the function. What happens if you need to make a change to the Spaces function? You will have to locate all the programs that use the function and then make the same change in all of these programs - tedious.

Why not put the Spaces function into a unit? Doing so will make the future changes to the function much easier since the only place you will have to make the changes is in the unit. So, open a new editor window and enter the following, which is a unit called Util, containing the Spaces function:



Name your unit `Util.pas` by choosing Save or Save As... on the Project menu. You have created the source of a unit, let us use it in the following program:

```
Program Fourth;
Uses Util;
Begin
  WriteLn(Spaces(30), 'Hello!');
End.
```

Save and name this program `Fourth.pas`. Since you have created a new unit which has not yet been compiled you should now choose Make or Build All from the Program menu before Run. This will make HighSpeed Pascal compile your `Util` unit as well as the program you just entered.

It's as simple as that.

That completes our quick tour of the HighSpeed Pascal integrated system; there is much more to learn but, hopefully, we have given you enough information for you to feel confident to explore the rest of the features using the manuals and an accompanying book. Remember that the Amiga is an advanced computer with a fairly complex operating system and you will need patience and determination - good luck.

Here is a brief summary of the commands available from the editor's menus - for complete information you should refer to the next two chapters.

The menus

Project

The Project menu contains commands which operate on the whole file being edited. This includes:

Creating new files
Loading existing files from disk
Saving the file
Reverting to an old version
Printing a project or part of it
Information about your file and HighSpeed Pascal
Quitting HighSpeed Pascal

Edit

This menu allows you to do things with a selected block of text. A block may be marked by clicking the mouse over some text and dragging over the window. The commands are:

Cut/Copy/Paste text using the clipboard
Erase text
Undo and undelete (for when you make a mistake)
Save a block of text to disk
Paste a file into the current project

Search

Commands which search through the text appear in this menu:

Find and replace occurrences of specific text
Go to a program line number
Set and return to up to 10 bookmarks

Window

Open a new window on the same source file
Cycle through all available windows
Arrange the windows neatly
Move a window behind or in front of all the others
Centre the window over the cursor

Program

All commands which compile, run and debug programs are located on the Program menu. Most of these are available as keyboard shortcuts using the Ctrl key and the first letter of the name. They include:

Compile and Run a program
Debug a program
Compile, Make or Build
Find a run-time error
Execute another program alongside HighSpeed Pascal

Macro

Macros are a facility for 'teaching' the editor a sequence of commands or keystrokes which can then be repeated any number of times. You may

Start recording a macro
Finish a macro
Replay the macro one or more times

Settings

Contains general settings that control how the system works. Some of the functions provided are:

Creating icons for saved files
Editor settings
Locating the tools on disk and when they are loaded
Selecting a font for editing
Save all settings to disk
Recall a previously saved settings file
Access to the Pascal.cfg *file containing compiler options*

Chapter 3
The Editor

Introduction

HighSpeed Pascal is a complete package which allows you to create, edit and debug Pascal programs on your Amiga® computer, either from a friendly integrated environment based on the editor or from the CLI or Shell.

This chapter looks at using the editor that is supplied as the heart of HighSpeed Pascal and aims to give you a introduction to most aspects of the package - it does not detail the compiler, debugger or other tools, nor the technical aspects of the compilation process; these are covered in later chapters.

You may wish to take advantage of our special offers on a range of technical books that you will find invaluable if you are a serious assembler language programmer; for details, see the order form included with this package.

The Editor

The editor supplied with HighSpeed Pascal is fully integrated with the system which means that you can develop programs in an intuitive and interactive manner, creating and editing your programs in the same environment as running and debugging your finished masterpiece.

Moreover, those of you with strong preferences for your own editor can dispense with the HiSoft editor and use your own favourite package along with the stand-alone version of HighSpeed Pascal, although you will lose the benefits of interactive development.

The editor for HighSpeed Pascal is a multi-window screen editor which allows you to enter and edit text and save and load from disk. It also lets you print some or all of your text, search and replace text patterns and set bookmarks throughout the text so that you can find key points in your program quickly. In addition, you can define and use macros.

The editor is Intuition-based, which means it uses all the user-friendly features of Amiga® programs that you have become familiar with such as windows, menus and mice. However, if you're a die-hard used to the hostile world of computers before the advent of WIMPs, you'll be pleased to know you can do practically everything you'll want to do from the keyboard without having to touch a mouse.

The editor is also 'RAM-based'; the file you are editing stays in memory for the whole time, so you don't have to wait while your disk grinds away loading different sections of the file as you edit. As all editing operations, including things like searching, are RAM-based they act extremely quickly. The file size is only limited by the amount of memory in your computer.

Once you have typed in your programs it is not much use if you are unable to save them to disk, so the editor has a comprehensive range of save and load options, allowing you to save all or part of the text and to load other files into the middle of the current one, for example.

To get things to happen in the editor, there are various methods available to you. Features may be accessed in one or more of the following ways:

- Using a single key, such as a Function or cursor key;
- Clicking on a menu item, such as Save;
- Using a menu shortcut, by pressing the right Amiga key in conjunction with another, such as Amiga F for *Find*;
- Using the Control key (subsequently referred to as Ctrl) or Alternate key (called Alt from now on) in conjunction with another, such as Alt-← for *cursor word left*;
- Clicking on the screen, such as in a scroll bar.

The menu shortcuts have been chosen to be, hopefully, easy to remember and to be compatible with many other Amiga® programs.

Two versions of the HighSpeed Pascal editor are supplied. One is specifically designed to take advantage of Release 2 of the operating system and its many new features. There is also a 1.3 version which gives you the same look and feel on an Amiga® running earlier versions of the operating system. Be sure to use the correct version for your machine.

Starting the Editor

From the Workbench simply double-click on the HSPascal icon to load it and display an empty window. If you have selected other icons as well as the editor icon (by Shift-clicking), these files will be opened as projects for editing when you run the editor.

To run the editor from the CLI, type its name (HSPascal) followed by an optional list of file pathnames - a file extension of .pas will be appended to filenames where necessary.

To find its startup settings, the editor searches for a file called HSPascal.prefs, firstly in the current directory then in the editor's directory and finally in ENV:HSPascal - if it cannot find this file it will use its built-in defaults. You may override this search from the Workbench by adding a SETTINGS=<pref_name> tooltype to a project icon or to the HSPascal icon.

Default icons may be provided by placing them in the ENV:HSPascal directory. An icon named def_pas will be used as the default for all files ending in .pas - if no such icon is found for a particular file type the def_project icon will be used instead.

We will return to a full description of projects, windows etc. after some general information about HighSpeed Pascal's use of gadgets and the like.

Requesters and gadgets

The editor makes extensive use of requesters which employ the Workbench 2 user interface (even if you are running under AmigaDOS 1.3) so it is worth recalling how to use these requesters and their associated gadgets. Note that most HighSpeed Pascal requesters do not restrict you from further editing.

If you require more detail than is given below, you should look in the *Amiga User Interface Style Guide* or the *Using the System Software* user guide supplied with your Amiga®.

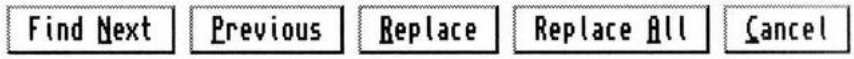
Keyboard shortcuts

Before we describe the different types of gadgets available, we should mention that most of them can be accessed either using the mouse or the keyboard. Keyboard shortcuts may be used whenever a text gadget is *not* active and the key that addresses the gadget is shown underlined in the descriptive text next to the particular gadget. For example, in the Find requester shown below, you can use N or n to find the next occurrence of the defined string or C (or c) to cancel the operation.

As we have said, the editor's requesters contain various gadgets of differing types and we shall now describe them.

Action gadgets or buttons

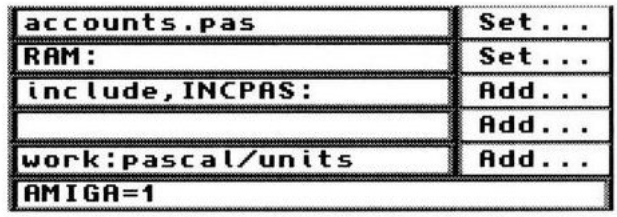
Action gadgets are boxes containing a description of the action that will take place should you left-click on the box or button.



action gadgets or buttons

To activate the action displayed by the button, point at it and left-click - the action will be triggered when you let go of the mouse button; this allows you to move off the gadget while still keeping the mouse button depressed if you change your mind.

Sometimes, clicking on a button will cause another requester or window to be displayed. If this is the case, the action gadget's text will end in an ellipsis (three periods):



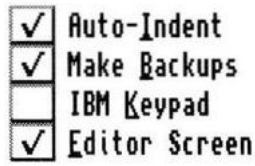
the Set... buttons lead to a file requester

The Cancel button will always be positioned at the bottom right of the requester with a keyboard shortcut of C and will abort the requester, leaving everything as it was before the requester was activated.

Many requesters also have a Use button which accepts any changes you have made and closes the requester. Clicking on a requester's close window gadget or pressing Esc has the same effect.

Check box gadgets

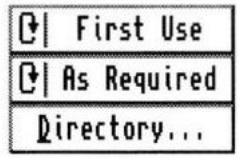
A check box is a small square that is either empty or contains a check mark or tick. This type of gadget allows you to turn some action on (checked) or off (blank) - simply click on it, with the left mouse button, to change its state.



check box gadgets with keyboard shortcuts

Cycle gadgets

Cycle gadgets allow you to choose one option from a list of several related items. To change to another option you must left-click anywhere in the gadget which will cycle forward through the list. If you hold the Shift key down while left-clicking, the list will cycle backwards. Here's an example:



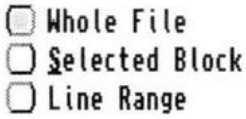
cycle gadgets allow selection from a list

As usual, you can use any keyboard shortcut instead of the mouse - again use Shift and the shortcut to go backwards through the list.

When you reach the end of the list, it will wrap round to the first item.

Radio buttons

Radio buttons are also used when you have to make a choice of one item from a short list of alternatives.

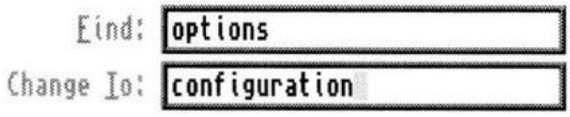


radio buttons used for a list

In the above example, you can choose to print the whole file, a marked block or a block defined by a line number range. When you choose one, the previous option becomes de-selected. Use Shift and the keyboard shortcut to move back through the list.

Text gadgets

Text gadgets allow you to enter strings of characters or numeric values.



text gadgets

To type into a text gadget you must activate it to obtain a cursor. Simply left-click inside the text gadget or press its keyboard shortcut to activate the gadget. Some commonly used requesters automatically activate a text gadget so that you can type directly into it as soon as the requester pops up.

As you are entering text, certain keys allow editing of the string:

←, →	move the cursor left or right through the text
Shift-←, Shift-→	move the cursor to the start of the line or to the end of the line
Backspace	delete the character behind the cursor
Shift-Backspace, Shift-Del	(release 2 only) delete from the cursor to the start or end of line.
Del	delete the character under the cursor
A X	delete the entire line
A Q	restore the text to its former state
Tab, Shift-Tab	(release 2 only) activate the next or previous text gadget
Return	exit the text gadget

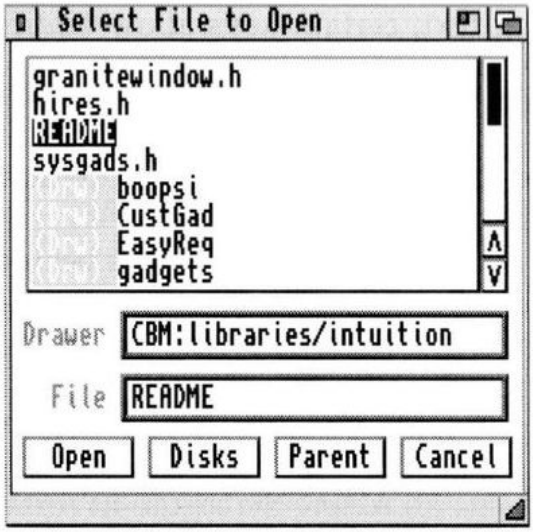
key commands available within text gadgets

To exit a text gadget simply press Return or left-click somewhere else in the requester.

File requesters

Whenever a file or directory name is needed, the HighSpeed Pascal editor will produce a File Requester which allows you to locate the desired object and select it. The Workbench 2 version uses the standard ASL file requester whilst the 1.3 version utilises the ARP requester.

The operation of each File Requester is similar and straightforward. However, a brief summary is given below.



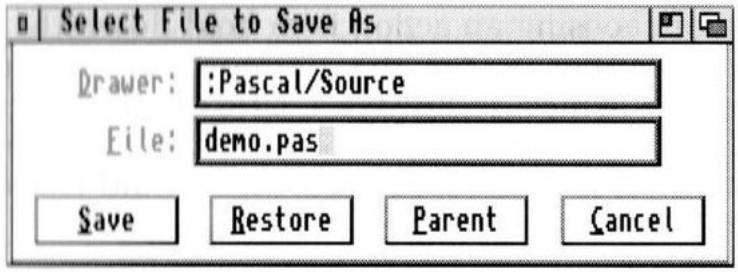
the ASL file requester

The requester title bar shows the type of object which you are required to select and the operation that will be performed on it. Save requesters appear in a different colour. By using the scroll gadgets you may move about the list of files and directories; clicking on a name will enter it into the appropriate text gadget. You may also choose to type into these gadgets to select a known file quickly.

The four action gadgets at the base of the requester allow you to proceed with the operation (this button will be labelled accordingly), show all available volume names and AmigaDOS assignments, move into the parent of the current drawer and cancel the operation (the window close gadget is also available for this purpose). Double-clicking over a name will both select the file and proceed, which normally passes the filename back to the calling program.

If the object you select does not exist or is of the wrong type you will be given a chance to reselect. Any errors are reported in another requester.

The ASL file requester also has the added benefit of a sizing gadget (the requester size and position will be remembered when you *Save Settings*) and a pull down menu containing useful Amiga® key shortcuts. When you are saving a file, typing a drawer name which does not exist gives you the chance to create that drawer.



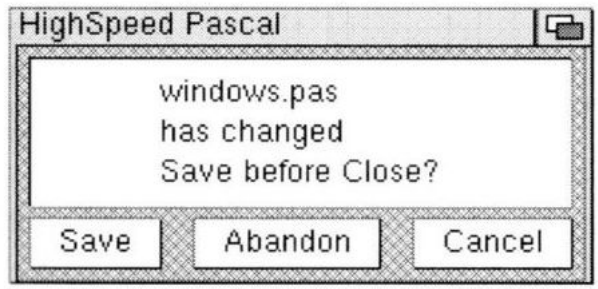
the simple File Requester

There is also a simple file requester which will appear in a number of situations. Firstly, if the editor is unable to open the ARP or ASL library (according to the version) from your LIBS: drawer, the simple requester will be used instead. It will also be used if there is insufficient free memory to open the full file requester. Note that, in extreme low memory situations, it may not be possible to open a requester at all, in which case you may need to close some unmodified projects or quit any other applications which are running.

Finally, you may obtain the above File Requester by holding down the Shift modifier when selecting a filing command from the menu or with the keyboard. For example, Shift-A will allow Save As... from the simple file requester. This can be useful if some other disk-intensive program is working and you do not wish to affect its performance.

Confirmation requesters

One other type of requester which you will see appears when it is necessary for you to make a choice before proceeding. Perhaps the most common occurrence of this is the modified project requester:



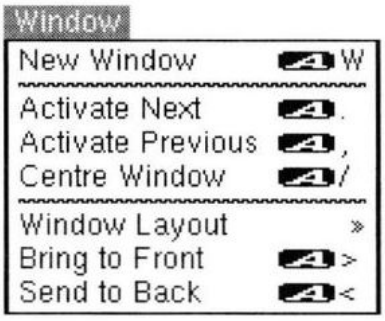
This is produced if you select some action which would lose any changes made to a project. By clicking on the gadgets or pressing the first letter of their titles you can choose to save the project, to abandon the changes and go ahead with the action or to back out, leaving everything as it was before you selected the action.

Another way of choosing an action is to hold down the left-Amiga key and press either V for the leftmost button or B for the rightmost button which is always Cancel. Remember that the continue action is placed at the left and Cancel at the right. The middle button often causes an action which would lose or overwrite some data.

Note that, unlike most HighSpeed Pascal requesters, these will block any further input to the editor until you have chosen an action.

Menus and sub-menus

Menus allow you to use editor commands quickly and easily while giving you the opportunity to browse through the various choices. To access the HighSpeed Pascal menus simply press the right mouse button and hold it down.

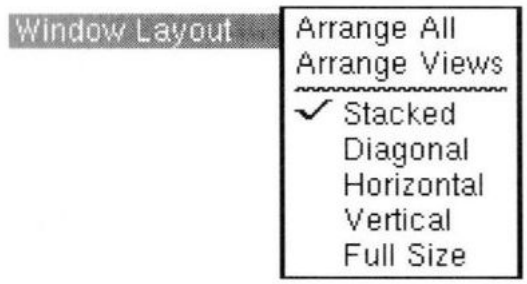


a typical HighSpeed Pascal menu

To select from the menu, still holding the right button down, move the pointer over one of the menu titles which appear at the top of the screen. Once the menu pops up, drag the selection bar to the required choice and let go of the mouse button. Many menu commands have keyboard shortcuts accessed by holding the right-Amiga key down and pressing the relevant command key as shown on the menu.

To select a sequence of menu items while still viewing the menu, keep your finger on the right mouse button and click on the *left* mouse button when you are over the relevant choice - you will then still be able to move up and down the menu and perhaps select another item.

Some options are accessed via 'sub-menus' which are shown by the » symbol to the right of the menu item; for example, if you move the mouse over the Window Layout selection on the Window menu, a sub-menu like this will appear:

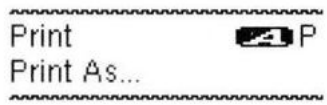


A sub-menu

You can then move the mouse to the right to select the particular item that you want. To cancel the operation just let go of the mouse without selecting a sub-item or move to another item on the main menu.

Some menu items can have a 'tick' or check mark next to them. Selecting such an item will select or de-select that choice. Once again you may use left-clicking to select a number of these options. The Window Layout sub-menu uses such items to give you a choice between the various layout styles.

If a menu item leads to a requester containing further choices, the text of the menu item will be followed by an ellipsis (three dots) as shown below:

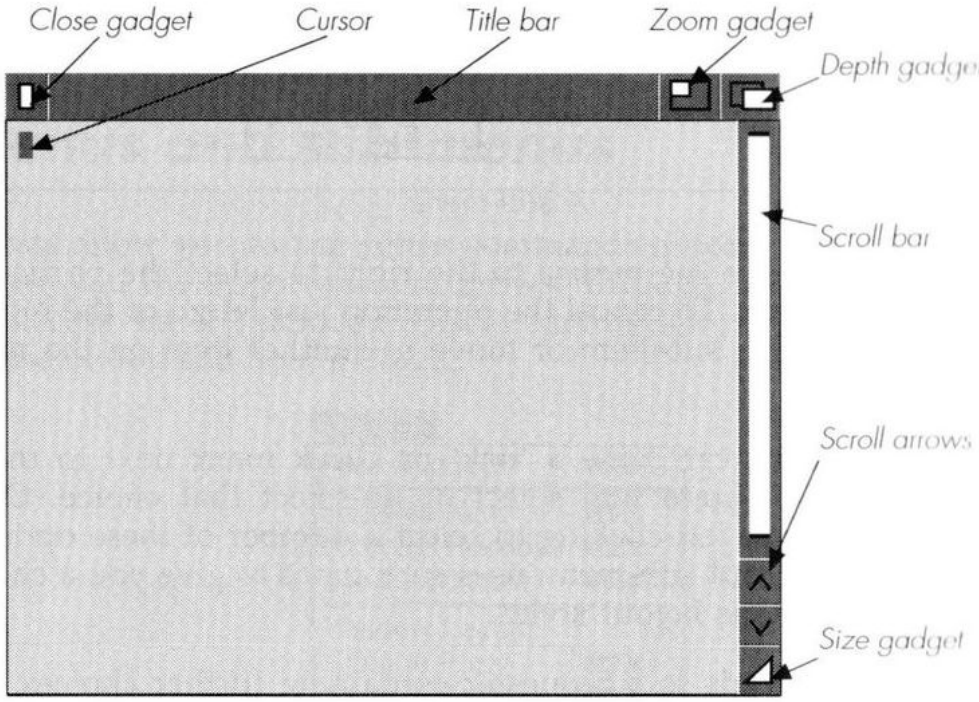


Print As... leads to a requester

The Editor's windows

Having loaded HighSpeed Pascal, you will be presented with an empty window and a graphic block, which is the *cursor*, in the top left-hand corner.

The window used by the editor is a standard Intuition window, so you can move it around by using the *Title bar* on the top of it, you can change its size by dragging on the *Sizing gadget*, close it with the *Close gadget*, bring it to the front or send it to the back with the *Depth gadget(s)* and make it full size (and back again) by clicking on the *Zoom gadget* (AmigaDOS 2 only). You can also move the view on the text by using the *Scroll gadgets*.

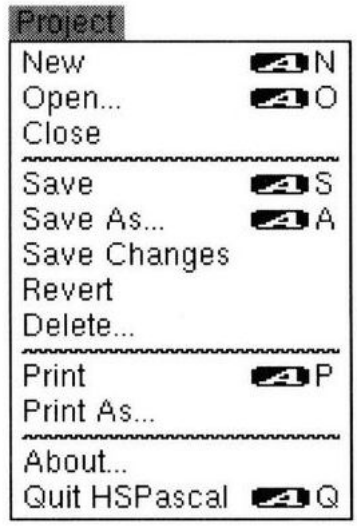


the HighSpeed Pascal window under AmigaDOS 2

Windows and Projects

We make the distinction between *windows* and *projects*: a project is a file with any number of windows open on it. You can also edit many different projects at any one time, each of which may have many windows.

To start a new project select New from the Project menu or type A N - this will give you an empty, untitled window. To open an existing project from disk you should use the Open command (A O) on the Project menu which will bring up a file requester allowing you to select a file to edit.



the Project menu

Once you have selected a file to edit you can open another window on that file by using ⌘W (New Window on the Window menu); subsequently close a window by left-clicking on its close gadget or by typing Shift-⌘W.

To finish a project (and close all of its windows), select Close from the Project menu, closing the last window on a project has the same effect. The other selections on the Project menu will be discussed below under the appropriate heading.

Switching Windows

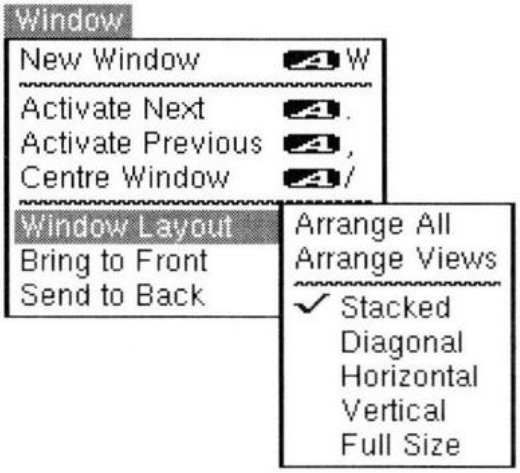
The editor has support for any number of windows, memory permitting, you can even have many windows (views) open on one file. Once you have some windows open, you can move between them in a variety of ways:

Firstly, you can select a window (if you can see it) by clicking on it with the mouse but this will not bring it to the front - you must use the window depth gadgets or Shift-⌘> to achieve that.

Under AmigaDOS 1.3 there are two window depth gadgets; one of these sends a window to the back and the other brings it to the front. We have implemented two keyboard shortcuts for these actions - A> (Shift-A .) for bring to front and A< (Shift-A ,) for send to back. With AmigaDOS 2 there is only one depth gadget which you can Shift-click to send the window to the back. These commands leave the window upon which the action was performed *selected*, so that, if you send a window to the back revealing another window, this new window will not be selected (and therefore you will not be able to type into it) until you click on it.

A useful action is to cycle through all the available editor windows using Activate Next and Activate Previous commands from the Window menu (the keyboard shortcuts are A . and A , respectively); these commands bring the relevant window to the front and select it, ready for use.

The windows can be organised in a number of ways and you can select this using Window Layout on the Window menu.



the Window Layout sub-menu

First of all, choose which layout you would like;

Stacked re-sizes the windows to the same width and arranges them so that you can see the title bar of each window (depending on which window is at the front).

Diagonal arranges all the windows diagonally and changes their size so that you can see the top right area of all the windows (and hence their depth gadgets).

Horizontal organises the windows one above the other and resizes them so that you can see the whole width of every window - this is most useful for a small number of windows.

Vertical is like Horizontal except that the windows are arranged side by side.

Full Size means that each window is made the size of whole screen and placed one behind the other.

Having chosen the desired arrangement (this will not actually change any windows) you can then Arrange All windows that are open or only Arrange Views - this means only arrange the windows associated with the current project.

Try this out for yourself to get the idea of how the different arrangements work. The sub-menu works best if you left-click over a layout style then release the right mouse button over one of the Arrange items.

Entering text and moving the cursor

To enter text, simply type on the keyboard and at the end of each line press the Return key (or the Enter key on the numeric pad) to start the next line. You can correct your mistakes by pressing the Backspace key, which deletes the character to the left of the cursor, or the Delete key, which removes the character under the cursor.

Undo Line

The HighSpeed Pascal editor keeps a copy of the line that you are working on so that as long as you have not moved off the current line, you can undo the changes you have made and recover the line as it was before you started editing it.

To restore the original line select Undo Line from the Edit menu or press ⌘Z. Remember, this will always work as long as the cursor is still within the line that you wish to recover.

Cursor keys and the mouse

You use the cursor keys, together with various *keyboard modifiers*, to move quickly around the file without needing to touch the mouse. The modifiers used are Shift, Ctrl and Alt and their use conforms to the guidelines given in the *Amiga User Interface Style Guide* which are as follows:

Using the cursor keys by themselves will move the text cursor around the file by a character (← →) or a line (↑ ↓) at a time, scrolling the display where necessary.

Holding the Shift key down while using the cursor keys will move to the edge of the *window* (i,e, the top/bottom/left/right) or show a new windowful of text if the cursor is already at the extreme of the window. Shift-← or Shift-→ have the effect of moving to the beginning or end of the current line. Shift also modifies the Backspace and Del keys in a similar way.

Using the Ctrl key with the cursor keys moves to the appropriate extreme of the *project* i.e. the beginning or end of line (Ctrl-← or Ctrl-→) and the top or bottom of the file (Ctrl-↑ and Ctrl-↓).

Finally the Alt modifier works on *words* so that Alt-← or Alt-→ move the cursor a word left and a word right respectively whilst Alt-↑ and Alt-↓ move on a page basis (like Shift). When used with Backspace or Del, Alt has the effect of deleting a word at a time.

The function of the Shift and Alt modifiers can be swapped (see *Settings*); this maintains some compatibility with previous HiSoft editors. It is also possible to set up the numeric keypad to act like an IBM PC keypad so that you can perform many cursor operations using its keys - see the *Settings* section for details.

You can use the mouse to move the cursor to a specific position on the screen by pointing and clicking. As usual, you can use the slider gadget to scroll through the file, in which case the cursor will move with the text. You can scroll the window up and down on a line basis by clicking on the vertical scroll gadgets.

If you position the cursor past the right-hand end of the line and type some text at that point the editor will automatically add the text to the real end of the line. If you type in long lines the window display will scroll sideways if required.

All the above commands along with the other keyboard shortcuts are summarised at the end of this chapter.

Tab key

The Tab key inserts a special character (ASCII code 9) into the text, which on the screen looks like a number of spaces, but is rather different. Pressing Tab aligns the cursor onto the next 'multiple of 8' column (by default), so if you press it at the start of a line (column 1) the cursor moves to the next multiple of 8, +1, which is column 9.

Tabs are very useful indeed for making items line up vertically and their main use in HighSpeed Pascal is for such things as indenting structured program lines. When you delete a tab the line closes up as if a number of spaces had been removed. The advantage of tabs is that they take up only 1 byte of memory, but can show on screen as many more.

You can change the tab size for each individual file - see *Settings* for details.

Return key

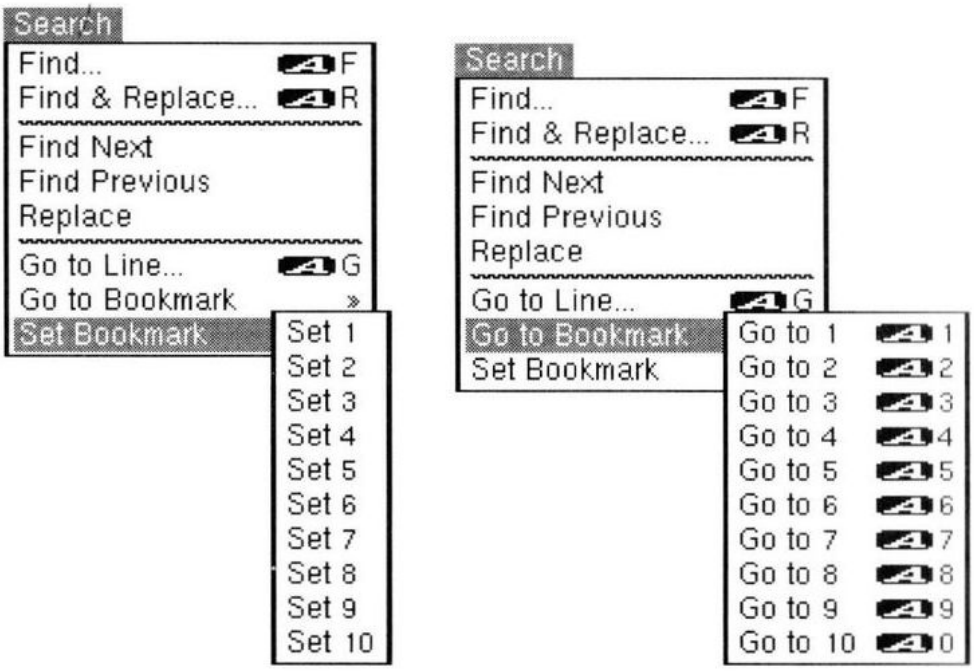
In addition to entering a line of text, the return key can be used to insert a blank line into the text by pressing Ctrl-Return.

When Return is pressed in the centre of a line, the line will be split into two. Ctrl-- (hyphen) can also be used to split a line at the cursor position whilst Ctrl-= will join the next line to the end of the current one.

Bookmarks

A further way to navigate your text is through the use of bookmarks. A bookmark is set by selecting the appropriate item on the Set Bookmark sub-menu from the Search menu or by using Shift-A and one of the digit keys on the top row of the keyboard, zero for 10 (e.g. Shift-A 4 sets bookmark 4).

Having set a bookmark in you are then able to select the corresponding item on the Goto Bookmark sub-menu to return the cursor to the line on which you set the bookmark; or use A and the appropriate digit e.g. A 2.



A possible total of 10 bookmarks are allowed per project which are saved along with the project only if Create Icons? is selected. Bookmarks are lost if the line they are on is deleted.

Deleting text

Backspace key

The Backspace key (to the left of the Del key) removes text to the left of the cursor. When pressed on its own it will delete a single character but when used in conjunction with the Alt or Shift modifier keys, it will delete the previous word or to the start of the the line.

If you backspace off the very beginning of a line this will normally remove the 'invisible' return character and join the line to the end of the previous line (although you can change this behaviour using Set Settings...). Backspacing when the cursor is past the end of the line will move the cursor to the end of the line without deleting a character.

Del key

The Del key removes text to the right of the cursor. In a similar manner to Backspace, just pressing Del will delete the character under the cursor whilst using the modifiers Alt and Shift with Del will delete by word or to the end of the line respectively.

Del will join lines together if used at the end of a line, unless you have changed this behaviour using Set Settings.... Using Del when the cursor is past the end of the line will simply move the cursor to the end of the line.

Delete line

The current line can be deleted from the text by pressing Alt-Backspace, Ctrl-Backspace or Ctrl-Y. The deleted line is remembered and may be later inserted back into the text.

Undelete Line

When a line is deleted using the above command it is preserved in an internal buffer, and can be re-inserted into the text by selecting Undelete Line from the Edit menu or pressing Ctrl-U. This can be done as many times as required which is particularly useful for repeating similar lines or swapping individual lines over.

Delete block

A marked block may be deleted from the text by pressing Alt Del or Ctrl-Del. These are equivalent to Erase on the Edit menu.

The following table summarises the behaviour of Backspace and Del when used with various modifier keys:

	Alt	Shift	Ctrl or Amiga
Backspace	delete previous word	delete to start of line	delete line (also Ctrl-Y)
Del	delete next word	delete to end of line	delete block

Remember that the current line is buffered, so you can undo all changes made to the line by using Alt Z Undo Line.

Block Commands

A *block* is a marked section of text which may be copied to another section, deleted, printed or saved onto disk. The editor will report `Block required` if a marked block is needed. Blocks may be marked using the mouse or with function keys.

Marking a block

The simplest way to mark a block is to left-click on the first character in the block and drag the mouse to the end of the block. The block is highlighted by showing the text in reverse as you drag the mouse. When you move the mouse past the edge of the window, the window will automatically scroll to allow blocks larger than the window size to be marked. You may mark a block by clicking at the end and dragging back if you wish.

Double-clicking will cause the word under the mouse to be marked as the block. If you double-click and then drag, text will be highlighted a word at a time.

A second way of marking a block is to place the cursor at one end of the block, either with the movement keys or by pointing and clicking. Then, holding down the Shift key, click at the other end of the block. This technique is often useful for marking a very large block as it can be used to extend an existing block.

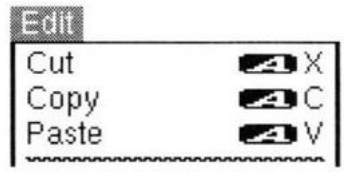
The start of a block may also be marked by moving the cursor to the required place and pressing key F1. The end of a block can be marked by moving the cursor and pressing key F2. The start and end of a block do not have to be marked in a specific order - if it is more convenient, you may mark the end of the block first.

To unmark any selected block, press Esc.

The Clipboard: Cut, Copy & Paste

HighSpeed Pascal supports the Amiga® clipboard, allowing you to not only cut and paste text within HighSpeed Pascal, but to share data between many different applications. The clipboard holds the most recently cut or copied block of text ready for subsequent pasting to another location.

Once you have marked a block you may copy it into the clipboard using Copy from the Edit menu. The text will remain in the file and the copy of the text held in the clipboard may then be inserted at another position by moving the cursor there and selecting Paste.



cut, copy and paste

The current block may be removed and place in the clipboard using Cut from the Edit menu; selecting Paste will then insert the block that was cut as before. To move a block of text involves marking the text, selecting Cut, moving to a new position and then Paste.

Cut block

A marked block may be deleted from the text by selecting Cut on the Edit menu. A deleted block is remembered in the clipboard for later use.

Copy block

The current marked block may be copied to the clipboard using Copy on the Edit menu or by pressing ⍓C. The block is automatically unmarked to show that it has been copied into the clipboard successfully.

This command can be very useful for moving blocks of text between different files by loading the first, marking a block, copying it to the clipboard then switching to another window or loading the other file and pasting the clipboard into it.

Paste block

The contents of the clipboard may be pasted at the cursor position by selecting Paste from the Edit menu or by pressing ⍓V. If the clipboard is empty or contains information other than text, the message `No text in clipboard` will be reported. If the editor is unable to access the clipboard, a Clipboard error is given.

Delete block

If you wish to delete a block without placing it in the clipboard, choose Erase on the Edit menu or press ⍓Del or Ctrl-Del. There is no way of retrieving text which is deleted in this way.

Save block

Once a block has been marked, it can be saved to disk by selecting Save Block... from the Edit menu. Assuming a valid block has been marked, a file requester will appear, allowing you to select a suitable drive, drawer and filename in which to save the block. You will be warned if you specify a filename which already exists in which case a backup will be made if requested in *Settings*.

To insert any previously saved file at the cursor position you may select Paste File on the Edit menu.

Printing a block

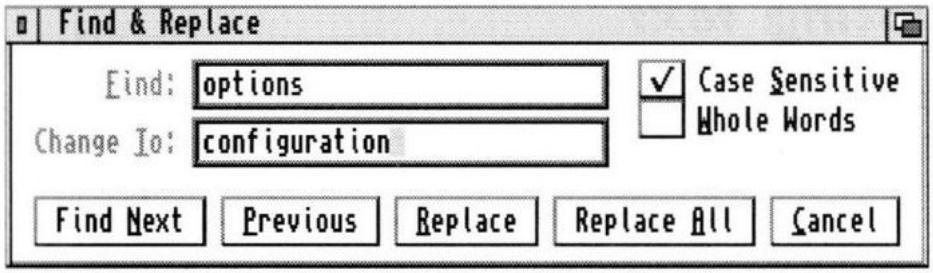
A marked block may be sent to the printer or a file by selecting Print As... on the Project menu. A requester will appear and you should choose Selected Block, the number of copies and the output name.

See the *Printing* section for more detail on the use of this requester.

Searching

The commands on the Search menu may be used for finding and perhaps replacing existing text. This is done by first selecting Find... (F) or Find & Replace... (R) to specify the search and replace text and then using either the requester or the Find Next, Find Previous and Replace commands.

The Find requester is a subset of the Find and Replace requester described below with facilities for searching only.



the Find & Replace requester

In the example above options has been entered as the find text and configuration as the text to replace it with.

If you click on Cancel, the requester will disappear, taking no action and restoring the previous find and replace text. If you click Find Next (or press Return) the search will start forwards, while clicking on Previous will start the search backwards.

If the search is successful, the window will be re-drawn with the cursor positioned at the start of the string. If the string could not be found, the message `Not found` will appear in the window title bar and the cursor will remain unmoved.

You may continue the search with Find Next and Find Previous or by activating an editor window and selecting Find Next or Find Previous from the Search menu. The keyboard shortcuts for these commands are Ctrl-N and Ctrl-P. Note that you do not have to close the requester in order to continue editing although this can be done via the close gadget or by pressing Esc.

Whether `test` is treated as the same as TEST or `Test` etc. depends on whether the Case Sensitive check box is selected. Normally, searching is *case insensitive*, meaning that upper and lower case letters are treated the same. However, with Case Sensitive, an exact match for the find text must be found before searching will stop.

Optionally, you may search for Whole Words by selecting the second check box. This will prevent a search for the word `in`, for example, matching the word `finish`. It will still find the word in *first-in-first-out* however because the hyphen has been used to separate the words.

Replacing text

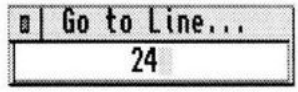
Having found an occurrence of the find text, it can be changed to the replace text by clicking on the Replace button. This is identical to selecting Replace from the Edit menu or pressing Ctrl-R whilst editing. Having replaced it, the editor will then search for further occurrences.

You may of course replace several occurrences of the text by selecting Replace several times in succession however, if you wish to replace every occurrence of the find string with the replace string starting from the cursor position, click on the Replace All gadget. This may take some time if there are a large number of occurrences. There is deliberately no alternative way of selecting Replace All in order to prevent it from being chosen accidentally.

You may search and replace Tab characters, line feed characters or any other control characters in a similar way by simply entering them in the appropriate text gadget. This facility can be used to search for something at the beginning or end of a line by including a line feed character (Ctrl-J) at the start or end of the find text. Note that AmigaDOS 2 users may have to hold down the right Amiga key while typing these or other control characters.

Go to line

To move the cursor to a specific line in the text, select Go to Line... from the Search menu, or press A G. A small requester will appear, allowing you to enter the line number and press Return.



The cursor will move to the specified line, centring the window over it. If the line does not exist, you will be positioned over the start or end of file. Pressing Return with no line number or clicking the close gadget will abort the operation.

Another fast way of moving around the file is by dragging the window scroll bar, which works in the usual fashion.

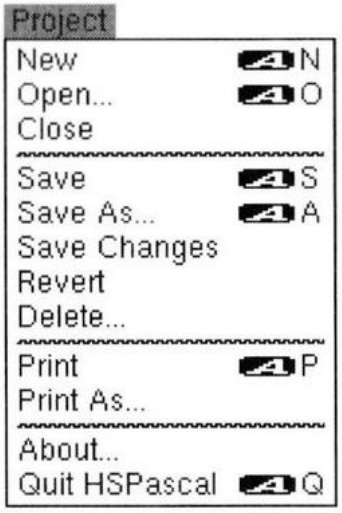
Go to block

The commands Shift-F1 and Shift-F2 take you to the start and end of any marked block, respectively.

Bookmarks

The Go to Bookmark sub-menu may be used to move to one of the project's 10 bookmarks previously set up via the Set Bookmark commands. Further details can be found in the *Bookmarks* section.

Disk Operations



The Project menu contains many operations that involve disk files; you can save and load your source file, insert text into your source, revert to the last saved version of your program, print your text, delete a file from a disk and more.

New

Select New to open an untitled project and window ready for entering text. The number of projects is limited by available memory only.

Opening files

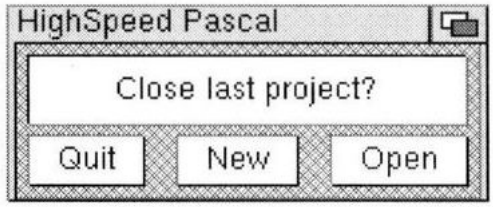
To load in a text file, select Open... from the Project menu, or press A O. This will open a new window if the current one is in use and then a file selector will appear, allowing you to specify the volume, drawer and filename. Assuming you do not Cancel, the editor will attempt to load the file. If it will fit, the file is loaded into memory and the window is re-drawn.

If an error occurs, a requester appears showing the error and filename, giving you a chance to Retry or Cancel as with all file operations. Some less common errors are reported as AmigaDOS error numbers as documented in your system manual.

A number of files may be opened automatically when HighSpeed Pascal is started by selecting project icons from the Workbench and Shift-double-clicking the editor icon, or by specifying a number of filenames on the command line.

Close

Closes the current project and all of its windows. This will also happen when you close the last window on a project either using the close gadget or by pressing Shift-⌘W. If the file being edited has changed since it was loaded or is a new file, you will be given a chance to save it before the window is closed.



Closing the last project will present you with the above requester which allows you to quit HighSpeed Pascal or open a new project.

Save

To save the text you are editing, select Save from the Project menu or press ⌘S. If the file was loaded from disk, it will be backed up (if requested in Settings) and the new version is written to disk. Untitled projects will produce a file requester, allowing you to select a name for the project like Save As....

If Create Icons? on the Settings menu is selected, the editor will attempt to generate a suitable Workbench icon for the file.

HighSpeed Pascal also saves the current tab size and positions of any bookmarks for the project in its icon. This information can only be saved to disk (and subsequently re-loaded) when Create Icons? is enabled.

Save As...

Save As... on the Project menu or ⌘A allows you to save a project in a different place or under a new name. It will be selected automatically if you Save an untitled project.

The File Selector will appear, giving the name and drawer where the file is currently located (if any). You may select the drawer and filename in the normal way. Clicking Save or pressing Return will then save the file onto disk. If you click on Cancel the text will not be saved.

If the filename you select already exists, a requester will appear asking you if you wish to overwrite this file, select a new name or cancel the operation. To save the file under the same name, use the Save command.

Backups and icons are also created if selected as described under *Settings*.

Save Changes

If you are editing a number of files and wish to save all of them to disk, Save Changes is equivalent to selecting Save for each modified file. You may use this before quitting the editor or for safety before running a program which you are testing.

Revert

Revert will warn you that you are about to lose the text in the selected window and, assuming that you choose to continue, will re-load the last saved version of the file you are editing.

Revert will do nothing if you try to use it on an unmodified file or a project that has not been saved previously.

Inserting a file

To read a file from disk and insert it into a project, select Paste File from the Edit menu. The File Selector will appear allowing you to select a filename or to cancel. The file will be read from disk and inserted, memory permitting, at the current cursor position.

Deleting files

You may want to delete a file from disk (if for instance you have run out of disk space whilst trying to save); click on Delete File. The File Selector will appear, allowing you to select a suitable disk and filename. Clicking OK or pressing Return will delete the file and its icon from disk and prompt for another one. Click on Cancel to stop (this will *not* cause any files to be deleted).

If an error occurs, a requester will appear showing an AmigaDOS error, the exact meaning of which can be found in your Amiga® manuals. Note that it is only possible to delete a drawer after any files which it contains have been deleted.

Quitting HighSpeed Pascal

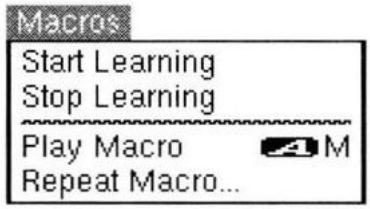
To leave HighSpeed Pascal, select Quit HSPascal from the Project menu or press A Q. If any changes have been made to the text and not saved to disk, a modified project requester will appear for each changed file giving you a chance to save or abandon the file. Selecting Cancel will abort the quit and resume editing.

If you have a large number of modified projects, all of which you wish to save to disk, it can be quicker to select Save Changes before quitting.

Macros

Macros provide a simple way of teaching the HighSpeed Pascal editor to perform a sequence of actions. This facility can be used for simple operations, such as indenting a number of lines by a single tab, or more complex ones such as adding a comment to the end of all lines containing a call to the procedure `CheckInput`.

The Macro menu gives access to the four commands used to record and replay macros.



the macro menu

To begin a macro you should select Start Learning from the Macro menu or press Ctrl-[.

The editor will subsequently remember each action you perform including all of the menu items, cursor movement commands and typed text. Mouse or requester operations are *not* remembered, only editor commands will be recorded.

One you have completed your chosen sequence of actions, select Stop Learning or press Ctrl-]. The macro can now be played back to repeat the recorded sequence.

Two playback commands are available. Selecting Play Macro or ⍙M will replay the macro once whilst Repeat Macro... or Shift-⍙M will play the macro several times in succession.

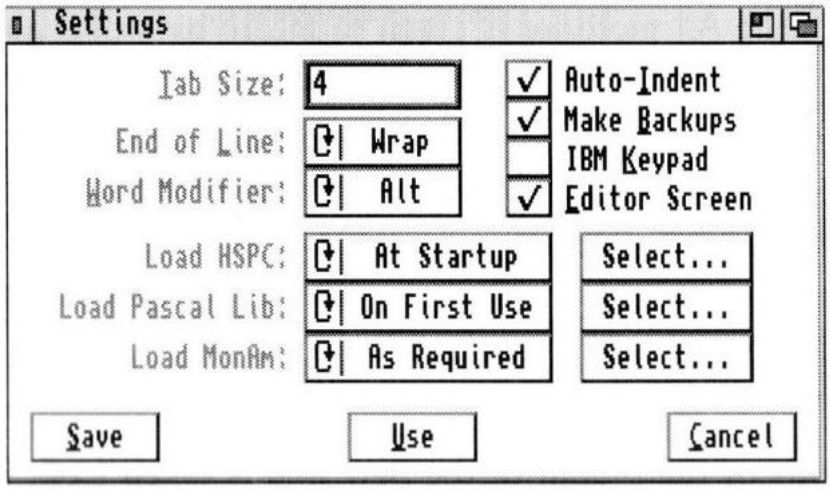
If you choose to repeat a macro, a number requester similar to the Go to Line requester will appear allowing you to enter the number of iterations and press Return. Clicking the close gadget or pressing Return without entering a number will cancel the operation.

The following sequence of actions would be required in order to record a macro which indented a number of lines by a single tab:

- select Start Learning (Ctrl-[).
- use Ctrl-← to move to the beginning of the line, press Tab and then cursor down a line.
- select Stop Learning (Ctrl-]) to complete the macro which will consists of these three actions.
- choose Repeat Macro and enter the number 10 to proceed to indent the next ten lines in exactly the same way.

Settings

Selecting Settings... from the Settings menu will produce a requester like this:



The editor preferences

This allows you to configure the editor as you like to use it; you can then save your customisation to disk so that the editor will always behave the same way. Selecting Use will accept any changes you have made without saving them to disk. Changes will be lost at the end of the current session unless subsequently saved.

Here are the different settings that you can change.

Tab Size

By default, the tab setting is 8, but this may be changed to any value from 1 to 20. This value will be used for the current project and as the default for any new projects. Tab sizes are saved for each individual project if Create Icons? is selected.

End of Line

There are three possible choices of the action that is taken when the editor reaches the end of a line. The default behaviour allows the most freedom although you can also choose for the cursor to wrap to the next line if you step past the end of a line or to treat the start and end line as terminators. This latter behaviour allows you to hold down the Del key to delete to the end of the line without it proceeding to delete the rest of your text.

The best way to find out which you prefer is to try using each setting.

Word modifier key

By default, the Alt modifier is taken to mean 'by word' and Shift is 'by line' as is standard for the Amiga®. However, users whose fingers are accustomed to other systems can swap this behaviour via the Word Key setting.

Auto-Indent lines

Selecting this option gives automatic line indenting. When active, an indent is added to the start of each new line created when you press Return or split a line.

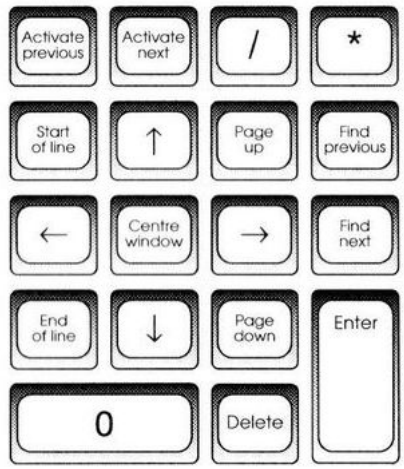
The contents of the indent of the new line is taken from the white space (i.e. tabs and/or spaces) at the start of the previous line. This allows you to lay out your program neatly by simply pressing Return.

Make Backups

Selecting this option causes the editor to make a backup (by adding the extension .BAK) when saving text files.

IBM Keypad

This option enables the use of the numeric keypad in an IBM PC-like way by default allowing single key presses for cursor functions. The keypad works as shown in the diagram below:



When this option is not selected, the keyboard reverts to returning the digits etc. although these functions may still be accessed by using the Ctrl modifier in conjunction with the numeric pad keys.

At all times, the Shift modifier and the numeric pad cursor keys will scroll the window in the appropriate direction whilst keeping the cursor in the same position over the text.

Editor Screen

The HighSpeed Pascal editor may be run either from the Workbench screen or its own screen if preferred. Selecting Editor Screen and saving the settings will cause a new screen to be opened when the editor starts up.

The Workbench 2 version opens on a public screen named HSPascal.1 by default. You are free to use this screen for other programs or for opening Shells via the SCREEN keyword of the CON: handler.

Loading of tools

The last three options in this requester relate to the various elements of the HighSpeed Pascal package.

You can choose to load the compiler, the Pascal.lib library (consisting of any number of pre-compiled units) and the debugger either when you run the editor (At Startup) or when the particular tool is used for the first time (On First Use). If you are low on memory, the As Required option provides minimal memory usage, only loading the file when necessary and then unloading it from memory when you finish using it.

Each Select... gadget produces a file requester allowing you to locate the directory and relevant file. This is also displayed if the editor comes to load a tool and cannot find it.

One final way of controlling tools is to load them externally by use of the AmigaDOS RESIDENT command. This way, the program will stay resident even when you quit HighSpeed Pascal which will save on loading time if you have sufficient memory. To pre-load non-program files (such as include files or individual units) we recommend copying them onto the RAM disk.

If you do not wish any Pascal.lib to be loaded when compiling, you may leave its selection empty.

Compile to Disk

When using the HighSpeed Pascal compiler from within the integrated environment, you may choose to generate programs in memory for speed or on disk using the Compile to Disk? option. This is described in more detail in the chapter on *The Compiler*.

Create Icons

With Create Icons? on the Settings menu selected, the editor will attempt to generate a suitable Workbench project icon for any saved file. The icon image is taken from the ENV:HSPascal or ENV:Sys directory and will match the file extension where possible. For example, the icon def_pas would be used for the files amiga.pas. If these could not be found then the icon def_project would be used instead.

You may create these icons with the Workbench icon editor and include a suitable default tool (this can be used to automatically run the editor when you double-click the icon) and other tooltypes as desired. Some common icon names are def_pas and def_i for source files with def_project for other files, def_prefs for settings files and def_tool for compiled programs.

Saving settings

To save the settings file you can choose Save Settings or Save Settings As... from the Settings menu. This latter command saves all editor preferences to the current settings file or creates a default file if no preferences have been loaded. Selecting the Save gadget on the Settings requester has the same effect. An icon of type def_prefs will also be created if you have selected Create Icons?.

In addition to saving the editor configuration, the current font, window layout style, find and print settings etc. are also saved. The current window position and size are saved as defaults for new projects and the relative position of all requesters are remembered.

The settings file itself is formatted in a similar way to Workbench icon tooltypes as a list of keywords in upper case followed by an equals sign and their associated setting. Advanced users may wish to modify the files textually within the editor in order to override only certain defaults or to specify an exact window position etc.

Note: although all of the settings described in this section are saved in the editor settings file, the information contained in the Pascal Config requester (described elsewhere) is *not* saved. To save and load configuration files you must use the menu on the Pascal Config requester itself.

Multiple settings files

The usual name for settings files is HSPascal.prefs. If you want to call your settings file a different name, you can use Save Settings As... which will allow selection of a name and drawer for the file via the File Requester.

When the editor is loaded, it looks for the HSPascal.prefs configuration file firstly in the current directory (which is the project icon drawer when started from Workbench by double-clicking), then in the editor directory (used by Save Settings when no file is available) and finally in ENV:HSPascal (for the convenience of network users). This can be overridden by the use of the SETTINGS=<file> tooltype which can be added to editor or project icons via the Workbench Info or Information option.

You can take advantage of this if you intend working on several projects, each requiring different settings, by saving settings files into each project drawer and a default settings file in the editor or ENV:HSPascal directory.

Loading settings

You may use Load Settings... from the Settings menu to select and load editor settings files from the File Requester. Workbench 2 users may wish to keep a number of settings files in the SYS:Prefs/Presets drawer.

Miscellaneous Commands

The About box

It you select About... from the Project menu (also available via the Help key), a requester will appear giving various details about HighSpeed Pascal, including its version number. You will also be told the name of the current project and its size both in lines and bytes.

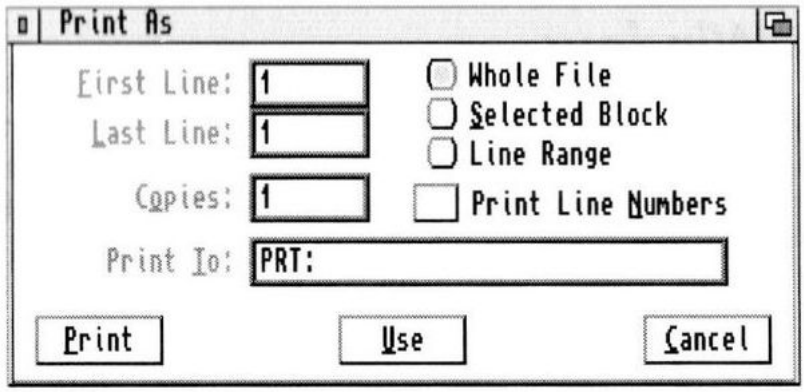
Pressing C or clicking on Continue will resume editing.

Printing

A range of printing options are available from HighSpeed Pascal.

To simply print a project using the current settings you may press A P or select Print from the Project menu. A requester will appear if the printer cannot be found or an error occurs.

You may change the printer settings from the requester below which is accessed by selecting Print As... or pressing Shift-A P.



the printer settings

Using the radio buttons, you may choose to print the entire project, the currently marked block or a selected range of lines entered via the first two number gadgets. Optional line numbers may be added and the number of copies specified. Multiple copies will be separated by a form feed and tabs are expanded to the appropriate number of spaces.

The Print To gadget may be used to specify where the listing is sent. For most uses this should be set to PRT: which is the default printer although other AmigaDOS devices may be used. In order to select the type of printer or the serial or parallel port, do this via system preferences in the normal way (see your Amiga® documentation for further details).

It is possible to specify a file or pathname in order to re-direct printing to disk. This can be used as a method of converting tabs to spaces or adding line numbers to a file.

One novel use of the Print As requester is to redirect printing of the selected block to the SPEAK: device. Marking a block and selecting Print will then proceed to 'read' the selected text using the Amiga's text-to-speech facilities!

Read-Only projects

This feature allows a project to be temporarily locked, preventing any accidental modifications. Blocks may still be marked or copied into the clipboard and the file can be saved but any command which would change the text will simply cause a `Project is Read-Only` error. Use Ctrl-W to turn this on or off for the current project.

Centre Window

Scrolls the current window so that the cursor is positioned as centrally as possible. Many commands such as Go to Bookmark or Find Next do this automatically although it is often a useful way of finding the cursor position when you get confused. The keyboard shortcut is A / or Ctrl-5 or Shift-5 on the numeric keypad.

Select Font

By default, all windows use the currently selected system font. For Workbench 1.3 this is normally topaz 8 or 9 as specified in *preferences*. Workbench 2 may additionally set up any fixed width font from the *Font preferences* editor.

The Select Font... command on the Settings menu gives you the ability to choose a separate font for the HighSpeed Pascal editor. A requester will appear and selecting a font and size followed by OK will adjust all project windows to use your chosen font.

Keyboard command summary

Project commands

⌘N	New
⌘O	Open
⌘S	Save
⌘A	Save As
⌘P	Print
Shift-⌘P	Print As
Help	About
⌘Q	Quit HighSpeed Pascal

Window commands

⌘W	New Window
Shift-⌘W	Close Window
⌘.	Activate Next
⌘,	Activate Previous
Shift-⌘>	Bring to Front
Shift-⌘<	Send to Back

Movement commands

← →	Character Left/Right
↑ ↓	Line Up/Down
Shift-←/→	Shift Left/Right
Shift-↑/↓	Shift Up/Down
Alt-←/→	Word Left/Right
Alt-↑/↓	Shift Up/Down
Ctrl-←/→	Start/End of Line
Ctrl-↑/↓	Top/Bottom of File
Shift-Numeric ←/→	Scroll Left/Right
Shift-Numeric ↑/↓	Scroll Up/Down
⌘/, Shift-Numeric 5	Centre Window

Edit commands

Tab	Insert Tab
Return	Enter Line
Ctrl-Return	Insert Line
Ctrl--	Join Lines
Ctrl-=	Split Line
Backspace/Del	Delete Character Left/Right
Shift-Backspace/Del	Delete to Start/End of Line
Alt-Backspace/Del	Delete Previous/Next Word
⌥Backspace, Ctrl-Backspace, Ctrl-Y	Delete Line
Ctrl-U	Undelete Line
⌥Z	Undo Line
Ctrl-W	Read-Only mode

Block commands

⌥X	Cut
⌥C	Copy
⌥V	Paste
⌥Del Ctrl-Del	Erase
Esc	Unmark Block
F1	Mark Start
F2	Mark End
Shift-F1	Locate Start
Shift-F2	Locate End

Search commands

⌥F	Find
⌥R	Find & Replace
Ctrl-N	Find Next
Ctrl-P	Find Previous
Ctrl-R	Replace
⌥G	Go to Line
⌥0-9	Go to Bookmark
Shift-⌥0-9	Set Bookmark

Macro commands

Ctrl-[	Start Learning
Ctrl-]	Stop Learning
⌥M	Play Macro
Shift-⌥M	Repeat Macro

Program commands

Ctrl-X	Run
Ctrl-A	Arguments
Ctrl-C	Compile
Ctrl-M	Make
Ctrl-B	Build All
Ctrl-F	Find Runtime Error
Ctrl-O	Compiler Options
Ctrl-S	Editor Settings
⌥E	Execute

Chapter 4
The Compiler

There are two methods of compiling programs with HighSpeed Pascal. Firstly, you may use the integrated environment provided by the HSPascal editor. This gives the benefits of quick compilation (with both source and object files residing in memory, space permitting) and integration with the debugger.

Alternatively, users who wish to run the compiler from a command line environment or script file have the option of using the Pascal compiler stand alone. This method is described in the second half of the chapter.

Compiling from the editor

Having produced your Pascal program using the editor and configured your compiler settings, you can then compile your program to memory or to disk and run it. All program operations can be found on the Program menu with keyboard shortcuts using the Ctrl modifier key.

Overview

Although each command is covered in greater detail later in this chapter, here is an overview of the development process:

- Set up any options you wish to use via the Pascal Config... entry on the Settings menu.
- If you wish to produce a stand-alone program on disk, select Compile to Disk also on the Settings menu. If you do not choose to compile to disk your program will be compiled directly to memory.
- Select the Run command (using Ctrl-X) to compile and run your program. If the editor cannot find the compiler (HSPC) you will be given a chance to locate it. This can also be done via the Settings requester.
- Fix any problems reported by the compiler and recompile. Once the program has compiled successfully, it will run automatically.
- If the program reported a run time error, select Find Error and enter the offset which was reported. The compiler will then locate the offending line and position the cursor over it for editing.
- If the program did not function as intended, you may want to select the Debug (Ctrl-D) option in order to enter the MonAm debugger and trace through the program until you locate the problem.

Command line arguments for program testing may be set up via the Arguments... command.

Remember that you may specify a particular file as your *primary source file* from the Pascal Config requester. This file (which may reside on disk) will be compiled instead of the current project to allow development of complex, multi-unit programs. Selecting Make as the default action will provide automatic remaking of any out of date units when using the Run command.

Program menu

The commands on the Program menu are used to communicate with the other parts of the package, the compiler, the debugger and, not least, the program that you are developing.

Compiler options may be set up through the Pascal Config... option on the Settings menu described in detail later in this chapter. You should be aware that the commands on the Program menu operate on the *primary file* found in this requester. If no such file is selected then the current project is used instead (this is the default behaviour).

Note that the compiler automatically reads include and unit source files from memory if they are loaded at the time - there is no need to save any changes to source files that are being edited before compilation since they are read from memory, not disk. Of course, files not loaded will be read from disk as normal.

A detailed explanation of each command follows.

Run

The command Run (keyboard shortcut Ctrl-X) recompiles your program if necessary and then executes the program. Any arguments entered via the Arguments... command will be passed to the executed program as a command line. The `System` unit `RunFromMemory` flag is also set for enabling special debugging sequences in your code etc.

Warning: if you think there is any chance that your program will crash the machine, make sure that you save your source code (Save Changes, Project menu) before selecting Run.

If the program has already been compiled, Run will execute it immediately. However, if it detects that your program source has changed since it was last run or compiled, the action selected in Pascal Config is carried out. By default this is Compile although you may change it to Make or Build All. For further details, refer to the relevant command.

When Compile to Disk is checked on the Settings menu, the Run command will read your program from disk. Although compiling to memory is faster, it uses more RAM and the executable program is lost whenever you leave the editor.

Any run time errors (as opposed to compile time errors) which occur will produce an error number and a program offset. This offset should be supplied (by you) to the Find Error... command in order to locate the corresponding source line. Note that crashed or looping Pascal programs will prevent the editor from unloading when you come to quit although you may still continue editing whilst programs are running.

When your program finishes executing you can close any window that it opened by hitting any key. Under Workbench 2 you can use Ctrl-\ to close the window.

Arguments

This command (Ctrl-A) lets you set up the command line that is passed to your program when it is run or debugged. You may access these arguments from within your Pascal program through use of the `ParamStr` and `ParamCount` functions in the `System` unit. The command line is saved within the editor settings file.

Debug

The Debug command (Ctrl-D from the keyboard) invokes the MonAm debugger which will load your program automatically, from disk or memory depending on your setting of the Compile to Disk item on the Settings menu. Your program will be stopped at the first instruction, ready to run or single step as required.

If the source has changed since the last compilation, Debug will automatically recompile in the same way as Run. Any arguments specified will also be passed to the program.

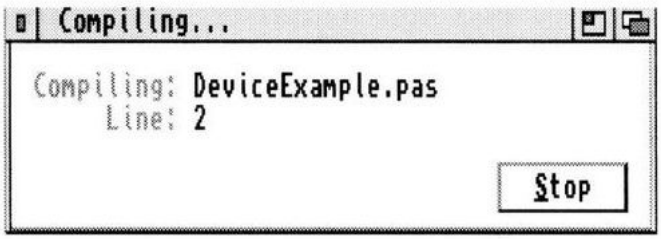
Note that when using the source level debugging facilities of MonAm it is often a good idea to select Save Changes before debugging. Not only does this protect you if the program crashes, it ensures that the correct versions of source files including any recent changes are loaded from the debugger for line number tracking. You may of course modify the source files from the editor if bugs are discovered whilst debugging.

For full details of using MonAm with Pascal programs, please refer to the *Debugger* chapter.

Compile

This command (keyboard shortcut Ctrl-C) compiles the primary or currently selected file without running the program. You may compile an untitled project before saving the source file to disk if you wish. The resulting object file will be directed to disk or memory according to the state of the Compile to Disk option on the Settings menu. Although compilation to memory is faster, it does use up more RAM.

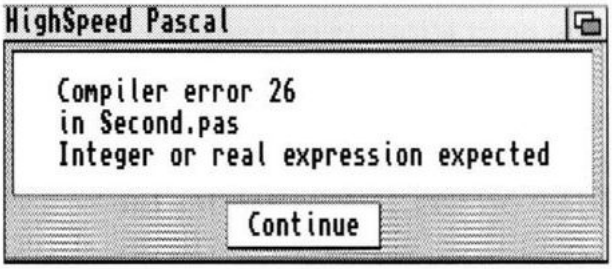
If the editor is unable to locate the HSPC compiler or Pascal.lib files you will be prompted with a requester. Choosing Select from this requester is identical to clicking on the relevant Select... button in the Settings requester. The location of the compiler and library may be permanently stored on disk via Save Settings.



Assuming that all goes well, you should see the above requester. The compiling requester informs you of the compiler's progress, i.e. the current file and line number. At any stage you may select the Stop button or close the window to abort compilation. Of course, if you do, no program will be produced.

Workbench icons are generated for program on disk if Create Icons? on the Settings menu is selected. The default program icon is named def_tool (see the Create Icons? description in the editor chapter for full details of how default icons are generated). Note that by default, no icons are generated for compiled units. You must provide a def_unit icon for these to be created.

If the compiler detects an error in your program or is unable to locate an include file or unit, compilation will stop and the error is reported in a requester similar to the one on the next page.



Where possible, the editor will load and activate the appropriate source file, marking the offending section as a block and positioning the cursor at its start. You should then correct the error and proceed to recompile. Once a program has been compiled successfully, you may test it with the Run and Debug commands.

Make

Make (Ctrl-M) is just like Compile except that it will automatically recompile any units which are out of date. This includes source files which have changed or are currently being edited, units which have not yet been compiled and old units compiled with a previous version of the compiler.

Using Make, you don't have to worry about which files you have changed and need to be compiled again - it does it all for you. This is especially useful when developing a large program comprising of many unit files in conjunction with the primary file facility. Make may be selected as the default action performed by Run via the Pascal Config requester.

For further details of the compilation process, refer to the Compile command.

Build All

This is used to unconditionally recompile every unit used by your program for which source is available. It is identical to selecting Compile for each unit that your program (or primary file) uses and then for the program itself. This is useful after selecting new compiler options in order to rebuild the entire program. The keyboard shortcut is Ctrl-B.

Find Error

Although this command is another way of invoking the compiler, it is not intended for producing a program at all. Instead, its function is to locate the source line which caused a run time error.

When a run time error is detected such as a range checking problem or divide by zero, the program will terminate and display the message

```
Runtime error XX at NNNNNNNN
```

where XX is the error number (as listed in the relevant Appendix in the *Technical Reference* manual) and NNNNNNNN is the program *offset*. This offset value should be entered into the Find Error requester before pressing Return to scan the program source looking for the statement which generated this code.

If the correct line is found, the cursor will be positioned over it in the normal way to allow correction of the problem. The keyboard shortcut for Find Error is Ctrl-F.

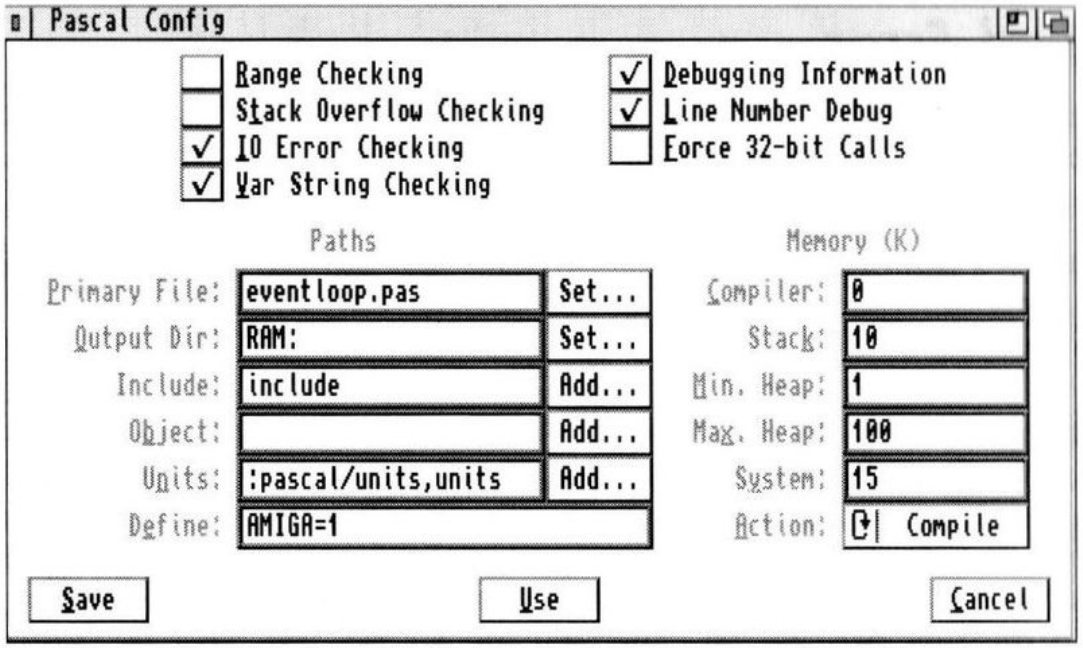
Execute

Execute (A E) displays a requester for you to enter a CLI command (and its arguments) to be executed. You may specify input and output redirection using the standard <, > and >> symbols followed by an AmigaDOS file or device name. Commands are multitasked with the editor so that you may continue to edit files or execute further commands whist it is running.

This facility can be used to display directories, copy files, format disks, run other programs etc. without the need to create an additional CLI or Shell. Of course if you wish to do this then you can always execute the NewShell command. The Execute requester's command is saved in the editor settings file.

Compiler options

The Pascal Config... entry on the Settings menu allows you to select the options used when compiling programs from the integrated environment. These options are identical to the ones used by the command line compiler and the editor can in fact save options files for use from the command line.



The Compiler Options

These options are described in greater detail under the command line options section but a summary of their use is given below.

Run time checks

There are 4 different types of run time checks which may be enabled. These will add to the size of your program but provide extra safeguards for the compiled program, detecting errors before they crash the machine or cause the program to behave incorrectly.

Range checking has two effects: it checks values assigned to simple types and ensures all array accesses are within the bounds of the array.

Stack overflow checking ensures that each procedure has sufficient stack space to function. If undetected, stack overflows cause memory corruption and general misbehaviour of your program. They can be caused by runaway programs, heavily nested procedures and use of recursion. One solution is to increase the size of the stack.

I/O error checking causes any input or output errors to terminate the program. If you are not using this you must explicitly test the `IOResult` after all I/O operations. Note that this only applies to the HighSpeed Pascal I/O functions, not to the Amiga operating system ones.

Switching off *VAR string* checking allows you to pass any length of string to a procedure which takes a string variable as a VAR parameter. You must ensure that strings are of sufficient size to store the data assigned to them.

Debugging information

There are two types of debugging information which can be produced by the compiler: symbols and line numbers. These are only of interest if you are going to be debugging your program and should be switched off when compiling the final version to disk or in order to save space.

Symbol debugging information enables you to use procedure names to set breakpoints and move around the code whilst line number debug permits tracking of your source files by the debugger. The Monam debugger supplied with HighSpeed Pascal understands both types of debug information.

Primary file

The primary file is a useful method of specifying which source file the program is contained within. Although for small programs this should be left blank in order to simply compile the current file from the editor, larger programs made up from a number of source files can be compiled without the need to continually switch back to the program source before selecting Compile.

You may select a primary file by typing in the text gadget or through a file requester produced by adjacent the Set... button. This will affect all commands on the Program menu which invoke the compiler and the current version of the file will be read from memory or disk as required.

Note that if the command line compiler finds a Pascal.cfg containing a primary file, it will automatically compile that file without requiring a source file on the command line.

Search paths

A further 4 options control where the compiler looks for and places certain types of files. As when locating the primary file, you may type into the text gadgets or use a file requester via the corresponding Set... or Add... button. Most of these search paths accept a list of directories separated by commas which will be searched in order.

The Output directory determines where compiled programs are placed instead of the current directory. The Include path is searched for files included with the $I directive. Object provides a the search path for the $L directive.

Finally, the Units search path is used when the compiler is looking for .unit files named in the USES clause in the source that are not present in any selected Pascal.lib file. Note that, in order to retain the correct organisation of disk files, the compiler will write any compiled units into the correct unit directory rather than the output directory specified for programs.

Define conditionals

HighSpeed Pascal provides control of conditional compilation through $DEFINE, $UNDEF, $IFDEF, $IFNDEF, $ELSE and $ENDIF sequences. You may also define conditional symbols through Pascal Config by providing a list of identifier names separated by commas.

Memory options

A number of memory options exist which allow you to configure compiler operation for your particular machine and Pascal program. All memory sizes are in Kb (Kilobytes).

The first option is the maximum amount of Workspace available for the *compiler*. It does not affect the resulting Pascal program in any way but allows you to limit the memory used when compiling or increase it to prevent out of memory errors during compilation.

The remainder of the memory options determine the memory configuration of compiled programs. Stack may be increased to avoid stack overflow problems and the default heap available for Pascal programs may be defined in terms of minimum and maximum sizes whilst always retaining a certain amount of memory to ensure correct functioning of the operating system.

Default Action

When using the integrated environment, the Run command on the Program menu will automatically recompile your program if it is out of date. Although this normally performs a Compile, you may select Make or Build All as the default action. Make is especially useful for multi-unit programs.

The default action also applies to the command line compiler, allowing you to generate Pascal.cfg files to carry out different actions.

Pascal Config files

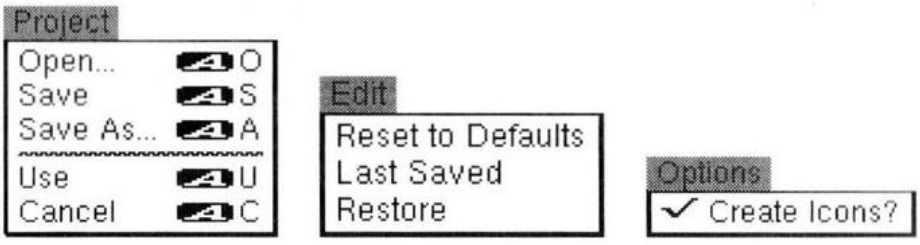
Unlike the rest of the HighSpeed Pascal settings, the contents of the Pascal Config requester are saved in a file separate to the editor settings. They are stored in a compiler options file called Pascal.cfg which may be loaded and saved in much the same way as normal settings files.

When the editor starts up, it searches for Pascal.cfg in the same places as it looks for HSPascal.prefs i.e. the project directory, the editor directory and ENV:HSPascal. This gives you the ability to save separate options files for different projects by placing them in the appropriate source drawer with a global default in ENV: or the editor drawer.

Config files have the additional benefit of being usable from a command line environment. HSPC will search for the Pascal.cfg file before processing its command line arguments allowing you to specify default options in this file. These may then be overridden by further options on the command line.

Even if you are not using the HighSpeed Pascal editor or integrated environment, you may wish to use it for generating Pascal.cfg files. This is done by selecting the requester described above and using the menu commands described below.

Menus in the Config requester



the Settings menus

Certain menus are available for use when the Pascal Config requester is displayed and active; these are shown above.

Those of you familiar with Workbench 2 will probably recognise the layout as that of a standard preferences editor. All of the menu commands work on the contents of the Pascal.cfg requester.

Selecting Open... allows you to recall a previously saved config file from disk. If you wish to revert to the default configuration, you may wish to use the Last Saved option instead. Last Saved checks in the current directory, ENV:HSPascal and the application directory in the same way as the editor.

Save (also available as a requester button) updates the current config file, storing any changes which you have made. If there is no current file then a default config file is created in the editor drawer. Save As... allows you to save the config somewhere else or under a different name although files named anything other than Pascal.cfg will not be loaded automatically by either the HighSpeed Pascal editor or the compiler.

Reset to Defaults may be used to restore all compiler options to their original state whilst Restore cancels all changes made to the requester since you started editing it.

Icons for config files are taken from def_cfg or def_project and will be saved only if Create Icons? on the Pascal Config menu is selected.

Compiling from the command line

The compiler may be used directly from the Shell or CLI. You can set up the compiler options to be used using the editor and then save them as described above. This will produce a Pascal.cfg file which is always read when the compiler is invoked by the CLI. The configuration file contains one command line option (as described below) per line.

The general format of the command line is

```
hspc {Options} FileName {Options}
```

The options are as follows:

-$R+	Enable range checking
-$S+	Enable stack checking
-$Mxxxx	Memory allocation parameters
-$F+	Force 32 bit calls
-$D+	Debugger symbols
-$L+	Line number debug
-$I-	No I/O error checking (default is on)
-$V-	No var -string check (default is on)
-Kxxx	Compiler's work space in K bytes
-Fxxx	Find runtime error
-B	Build all units
-M	Make modified units
-Q	Quiet compilation
-Dxxx	Define conditionals
-Exxx	Output directory
-Ixxx	Include directories
-Oxxx	Object directories
-Uxxx	Unit directories

The / character may be used to introduce options instead of - if you wish.

$ Switch Option

The -$ or /$ option allows you to enable or disable compiler directives, just as if they were entered into the first line of your program.

More than one can be entered at the same time by separating them with commas. Do not use any spaces. The -$M directive is followed by four numbers separated with commas. These numbers represent the number of K bytes requested for stack, heap min/max and free to the system.

Example:

```
HSPC -$R-,S+,M5,10,10,20 DEMO
```

Directory Options

Three of the directory lists entered can take more than one path. They are then separated by a semi-colon, comma or + sign.

The `-E` option defines where the generated units and programs are placed. This allows you to keep all units together or on a ram disk. Only one path can be entered.

The `-I` option defines a set of search paths for where to search for include files named in `$I` directives in the source.

The `-O` option defines a set of search paths for where to search for object files named in `$L` directives in the source.

The `-U` option defines a set of search paths for where to search for unit files units named in the `USES` clause in the source. If the `.unit` file corresponding to the main file is also found here, it is updated and any `-E` option is ignored.

Mode Options

The `-M` option forces the compiler to check all file dates to see if any units, include file or object file used has been changed. Each unit that then needs to be compiled will be compiled.

The `-B` option forces the compiler to re-compile all units used, if the source can be found.

If -M or -B is not used, the compiler does not check anything, it just compiles the file requested. If any units have been changed in the interface part, the compiler will complain.

The -Fxxx option will start the compiler in 'find run-time error' mode.

The -Q option will hide any output to the screen until an error occurs.

The -Dxxx option defines conditional compiler variables, used for source control. With this option you can make part of your code be compiled dependending on which flags (names) you activate.

The -Kxxx option specifies the number of kilobytes that the compiler will use for its workspace.

Compiler directives

HighSpeed Pascal can be controlled by using compiler directives, which are commands not defined in standard Pascal. Therefore they are placed in a special form of comment.

A directive must start with a $ dollar sign immediately after the starting bracket, i.e. {$ or (*$. The next characters are the command, which can be one or more letter(s). Multi-letter directives are used for conditional compilation.

The different directives have default values, either set by the command line version by the -$ commands given, or for the integrated version by filling the Compiler Settings requester.

Switch directives

Switch directives are written by using a single letter followed by a + or a -. The requester shows this as a check-mark next to the name of the option. A '+' (or a check-mark in the dialog) enables the feature.

Several options can be given in one comment as in (*$R+,S-*).

Debug Information {$D+}

Enables generation of debug information in the generated program file. The names saved are those for all procedures and functions used. This is called Debug Symbols in the requester.

Line debug {$L+}

Enables the generation of line number debugging information. This is output if this option is enabled when the `END.` of the program is compiled. The line number debugging information will increase the size of the file generated by 8 bytes per line that contains executable code. This is called Line Number Debug.

Far Calls {$F+}

Enables the program to be larger than 32 Kbyte. `$F` makes all procedure and function calls use the 32 bit version if needed. Each unit may need to be compiled with this flag. In Turbo Pascal for the PC, it is fatal to forget to use `$F`, because the information saved on the stack depends on the $F setting. On the 68000, the return address is always saved as 32 bit. The `$F` flag in HighSpeed Pascal only helps the compiler to make the best decision when making calls. This is called `Force 32 bit calls` in the Compiler Settings requester.

IO Error Checking {$I+}

enables automatic input/output checking, i.e. any IO error will terminate the program. If you use the `$I-` option, you will have to test the `IOResult` function yourself after each IO operation.

Range Checking {$R+}

Enables range checking on simple types. Every assignment to simple types is then checked, and every use of indexing an array or a string is also checked to be within its range.

Stack Checking {$S+}

Enables stack checking. Stack checking is performed at the very start of each procedure that is compiled using the $S+ flag.

VAR String checking {$V+}

Enables the (normal) var-string checking. Using the $V- flag you can pass strings of any length to a procedure that has a string variable as a VAR parameter. It is up to you to make sure that the string passed is not overwritten by more than it can hold.

Other single letter directives

Memory Size Layout

{$M stack,hmin,hmax,free}

Sets the default memory allocation. $M is ignored in a unit.

Each of the 4 parameters specifies a number of Kilobytes.

Stack is the request for stack space, minimum 1.

hmin is the minimum request for heap space, minimum 1.

hmax is the maximum request for heap space.

free is the amount of memory that is left free to the operating when the program is started.

Include Files {$I FileName}

Makes the compiler include the named file into the source. The file is searched for using the path(s) specified in the Compiler Settings requester or on the command line in the -I command.

Object Files {$L FileName}

Makes the compiler include the named object file when all declaration has been done. The EXTERNAL procedure and functions is assumed to be in a .o file. You can specify several $L directives, one for each object file. The object file is searched for by using the path(s) specified in the Compiler Settings requester or on the command line in the -O command.

Examples:

```
{$R-,S+,M 5,10,10,20}
{$S- Disable stack checking}
{$R+ Make sure range checking is enabled}
```

Source code control

The compiler maintains some variables that are not part of the program being compiled. They can be used to control the way the compiler works, by enabling or disabling the compilation of certain parts of the source.

The following 3 variables are defined in this version: `Amiga`, `68000` and `Ver15`. The last one is valid in this version release number 1.5. The variables do not contain any value, you just test if they are there or not.

New variables are defined with `DEFINE`, and removed by using the `UNDEF` directive.

The test for their presence is performed with either `IFDEF` (for IF DEFined then) or `IFNDEF` (for IF Not DEFined then).

The compiling state can be changed using the `ELSE` directive.

An `ENDIF` directive ends the `$IFxx` sequence.

Finally you can test if a switch option is set by using the `IFOPT` directive.

Examples of the use of conditional compilation.

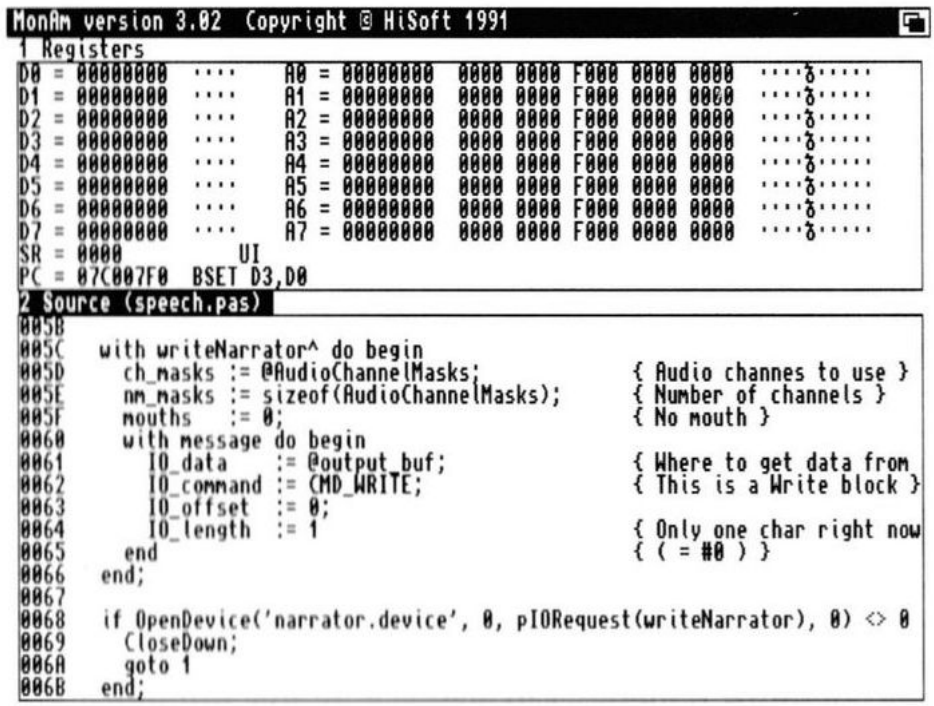
```
Unit OutStuff;
{$IFOPT R+}
 {$Define RangeCheck}
{$Else}
 {$UnDef RangeCheck}
{$EndIF}
begin

{$IfDef CPU86}
 N:=Swap(N); {Swap bytes if using the 80x86
family}
{$EndIF}

{$IfDef DebugMode}
 writeln('Debug=> N is now: ',N);
{$EndIF}
end.
```

Chapter 5
The Debugger

Introduction



The MonAm Debugger

Programming mistakes (known as *bugs*, after a spider that was found crawling around the core memory of one of the early computers) can range from the trivial, such as a missing CR in a printout, through the usual (an incorrect result) to the very serious where the computer crashes because you have used the wrong pointer or corrupted the system memory (like that spider).

To help you find and correct all forms of bugs, HighSpeed Pascal includes a debugger, MonAm. MonAm is a powerful symbolic debugger and disassembler which lets you examine programs and memory, execute programs an instruction at a time and trap processor exceptions caused by programmer error.

MonAm uses its own screen (in the Amiga® sense), so if you are debugging a program which uses windows, your program will not be sent re-draw messages or have its display destroyed when you single-step or breakpoint. This makes it particularly useful for graphical output programs such as applications running under Intuition.

If you are already an expert from using MonAm supplied with Devpac Amiga version 3, please read the next few pages on the use of MonAm with HighSpeed Pascal. Devpac 2 users should note that many new features have been added to version 3 of the debugger.

Debugging Pascal Programs

Although MonAm is a *low-level* debugger, displaying such things as 680x0 instructions and registers, it can be used to good effect for debugging programs written with any compiler that generates machine-code output such as HighSpeed Pascal. You will see procedure and function names within the code and you can view your original source and step through it by line etc.

However, in order to make use of the debugger and to understand this section of the manual you should have at least a basic knowledge of assembly language. It is not intended for the beginner.

In order to use the debugger fully with HighSpeed Pascal programs you must remember to enable both line number and symbol debugging information when compiling. Note that this does *not* give you access to local variables.

One thing to consider before groping around inside your program with MonAm is that many bugs can be discovered much more easily by simply looking at the source code. Take some time to explore the problem and then carefully study the parts of the program likely to produce such results. Using this method of debugging you are much more likely to spot structural errors or find a better way of doing something rather than simply patching up the problem.

Program initialisation

Before your Pascal program starts executing, a number of initialisation procedures will be called. These include the startup code which handles CLI or Workbench startup with parameters, HighSpeed Pascal initialisation and individual unit initialisation.

The first couple of instructions simply jump to the label `XProgram` which is the start of these initialisation procedures. They appear as a list of `JSR` instructions. Unit initialisation routines start at a label with the same name as the unit.

If you are not interested in the initialisation you can use the *Run Until* command and simply enter the program name to begin debugging. Alternatively you may wish to run until a certain line number in your source file.

Source files

MonAm is able to understand line number debugging information so that it 'knows' where in memory each program line starts and can automatically load your source code for you. Using these facilities, the debugger will 'track' the source code correctly showing both the current instruction in a disassembly and the corresponding line of Pascal.

By default, MonAm attempts to load the program source for you (although you may disable this via the *Preferences* command). For many programs, you need not worry about multiple source files but when working on a larger program which is divided into a number of units you will want access to each source file.

You may load further source files from disk by selecting the appropriate window and using the *Load ASCII* command (Ctrl-A). If any line number debugging was generated for the file it will also be 'locked' to the program counter so that it is tracked automatically. When control switches between source files (calling a procedure in another unit, for example) it is up to you to flip between them using the . and , keys.

When a source file is loaded, you may line numbers with all of the debugger commands through two special operators; # and ?. The first of these yields a program address, so placing a breakpoint at `#10` puts a breakpoint at line 10. The expression `#0` is taken to mean the first executable line of code. The second, less frequently used operator does the opposite, returning you the line number corresponding to a given program address.

Variables

Although there are no facilities for displaying a particular Pascal variable it is quite possible to examine or even change the values of program variable. All global variables are given a label name so that typing their name will return the *address* at which they are stored in memory, not their value. Setting a window start address to a variable name will allow you to view and edit that memory.

When interpreting the data you are looking at you must bear in mind the format in which the variable is stored, whether it is a byte, word or longword for example. An array is stored as a number of individual values, one for each combination of array indices. Strings are a length byte followed by ASCII characters. A record is stored with each field one after another. See the *Inside HighSpeed Pascal* Appendix in the *Technical Reference* manual for further details.

A quicker way to examine a numeric variable's value without upsetting your window display is to use the *Other Bases* command by simply pressing `0`. Something similar to the following line can then be entered:

```
{counter}.w
```

This would display the word (`.b` would show a byte whilst omitting the extension shows a longword) stored in `counter` in both decimal and hex.

Termination

You will normally want to run the program until you detect certain conditions or an exception occurs. In many cases you can just return control to the program with Ctrl-R and allow it to terminate normally. However, sometimes this is not possible.

If the program is about to do something wrong, such as pass the incorrect value or use an uninitialised pointer, you may be able to manually adjust the values so that it works correctly. If you want to abandon the program you can just quit MonAm which will leave the program in memory in a halted state.

If you wish to exit more cleanly then you should resume execution from the label `XHlt` which starts the chain of exit handlers and terminates the program. This may leave some resources lying around but it will terminate the program, allowing you to continue with further compilations etc.

Warning: be very wary of using the machine after a program has crashed. Any memory corruption can cause serious problems either immediately or much later on. If you think the machine may be in a compromised state it is always much safer to reboot before continuing.

MonAm Concepts

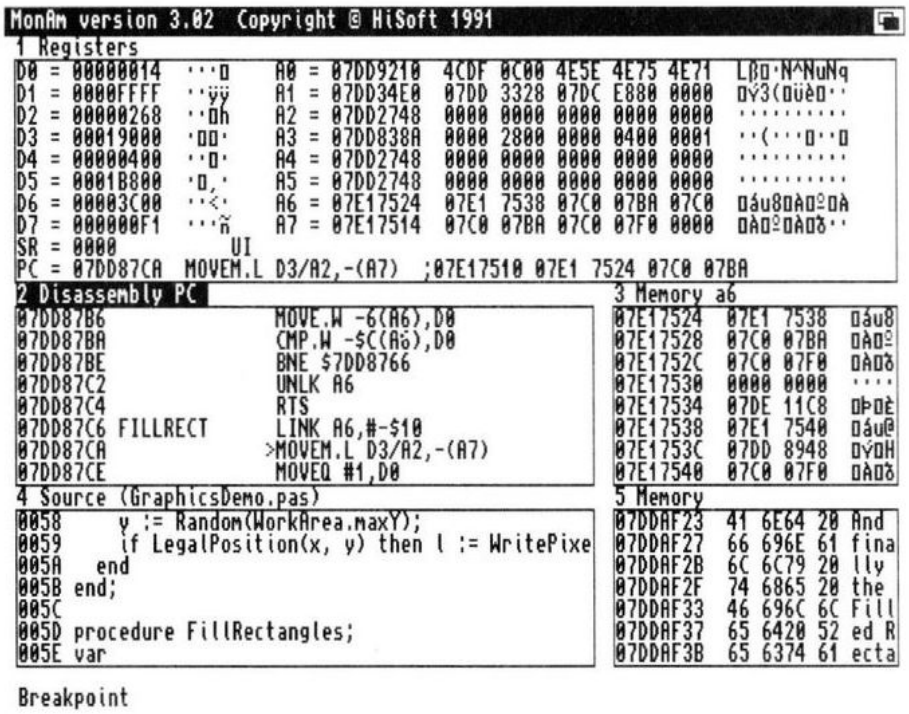
Here is a swift look at the concepts behind MonAm; it is a good idea to read this section before moving on to the next sections, even if you are an experienced programmer.

Front Panel Display

When MonAm is invoked it displays a *Front Panel* showing registers, memory, source code and instructions. The name Front Panel stems from the type of panels that were mounted on mainframe and mini computers to provide information on the state of the machine at a particular moment, usually through the use of flashing lights. These lights represent whether or not particular flip-flops (electronic switches) within the computer are open or closed; the flip-flops that are chosen to be shown on this panel are normally those that make up the internal registers and flags of the computer thus enabling programmers and engineers to observe what the computer is doing when running a program.

These were *hardware* front panel displays; what MonAm provides you with is a *software* front panel - the code within MonAm works out the state of the computer and then displays this information on the screen.

The MonAm display consists of a number of windows through which you can view the 680x0 registers, a disassembly of your program, your program's source code or a portion of memory - you choose what you want in each window (within certain limitations). The layout of MonAm's front panel is shown below.



MonAm's front panel

MonAm's Windows

As we have said, there are four different types of view through a window:

- a *register* window in which you can see the various 680x0 data and address registers, the program counter (PC), the status register (SR) and the current instruction. The values of the data and address registers are shown in hexadecimal together with some information about the locations to which the registers point.
- a *disassembly* window which shows a 680x0 disassembly of the memory that it is addressing, including any symbols that are found.
- a *memory* window which displays the contents of memory locations in hexadecimal and ASCII.

- a *source code* window. In this type of window you can view a text file which may be the source code of the program that you are debugging, assuming that this exists. You can display line numbers if you wish and, if the program that owns the source code has been compiled with line debugging enabled, you will be able to use this information to step through the program's source code and set breakpoints on source lines.

Up to five windows can be shown simultaneously or, by changing the width and height of the windows you, can show just two.

Each window is numbered from 1 to 5 and can display different types of information - window 1 can be of any type, register, memory, source code or disassembly; windows 2 and 4 can be memory, disassembly or source code windows whilst windows 3 and 5 are restricted to being memory windows.

Stacking Windows

Each window also has depth - you can stack views beneath a window so that you have almost limitless flexibility in what you choose to display.

In addition you can *split* and *widen* most windows; split means to grow or to shrink the window vertically whilst widen means to do the same horizontally. These operations may hide other windows temporarily or they may uncover hidden windows.

Locking Windows

Each window may also be locked to an arbitrary expression. Thus, you can lock a memory window to a register so that it displays the contents of the memory addressed by that register. Or you might want to lock a disassembly window to the `PC`, which is the default condition for window 2 unless you have saved.

Each view on the window stack can be locked to a different expression although it does not make sense to lock the register window.

All the above window features will be discussed in more detail later.

The Current Window

MonAm uses the concept of a *current window* - this is denoted by displaying its title highlighted and is the window on which any operation will take place.

The current window may be changed by pressing the Tab key to cycle between them, or by pressing the A key together with the window number, for example A 2 selects window number 2, even if it is hidden currently.

If your typing seems to be ignored in MonAm don't be alarmed; it means that another screen is active, such as that of your program. To correct this click on any part of the MonAm display. You can always tell when the MonAm display is active because the mouse pointer will be bug-shaped.

MonAm and Multi-Tasking

The Amiga® is a multi-tasking machine, and this imposes some restrictions on what MonAm can do. Having loaded a program (or task) into memory, that task is suspended. This means it is waiting, in this case for a MonAm command to let it continue.

The other state the task can be in is executing - running at the same time as MonAm. Some commands require one or other of these states to operate - for example you can only single-step a task that is suspended. If the task is running you will get the error `Task must be suspended!` when you try to single-step it.

MonAm can only debug one task at a time.

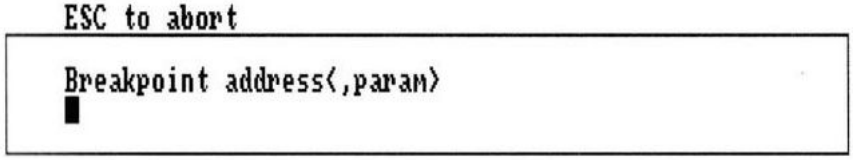
Symbolic Debugging

A major feature of MonAm is its ability to use symbols taken from the original program whilst debugging. MonAm uses standard AmigaDOS file `HUNK_SYMBOL` hunks as produced by most Amiga® programs that produce executable files, such as linkers, compilers and GenAm, the assembler supplied with Devpac Amiga. Naturally HighSpeed Pascal produces these.

MonAm can also accept line number information from various different types of HUNK_DEBUG hunk, which enables the debugger to handle source code files that are connected with the program being debugged on a line basis. If the program to be debugged contains this line number information, you will be able to set breakpoints in its source code and even single-step it, source line by source line. Products that support this, currently, are: HighSpeed Pascal, Devpac Amiga 3, HiSoft BASIC 2 and SAS/Lattice C 5.

MonAm Requesters

MonAm makes extensive use of requesters which are similar in concept to those in Intuition programs but have several differences.



a MonAm requester

A MonAm requester displays the prompt ESC to abort above the top left corner of the box together with a prompt, normally followed by a blank line or some text to edit, with a cursor. At any time a requester may be exited by pressing Esc, or data may be entered at the cursor by normal typing. Various keys may be used to edit the text:

←, →	move the cursor left or right through the text
Shift-←, Shift-→	move the cursor to the start of the line or to the end of the line
Backspace	delete the character behind the cursor
Del	delete the character under the cursor
A X	delete the entire line
Esc	abandon the requester

commands available within MonAm requesters

When you have finished entering a line, press the Return key; if the line contains errors the screen will flash and the Return key will be ignored allowing correction of the data before pressing Return again.

Some MonAm requesters simply display a message together with the prompt Return; these are normally used to inform you of some form of error. The box will disappear on pressing the Return or Esc keys, whichever you find more convenient.

Command Input

MonAm is controlled by single-key commands which gives a fast user-interface, although this can take getting used to if you are familiar with a line-oriented command interface of another debugger. Users of our debuggers on other machines such as the Atari ST/TT will find many commands are identical.

In general the Alt key is the window key - when used in conjunction with other keys it acts on the *current window*. The Ctrl key is usually used to invoke commands connected with execution of the program that is being debugged.

Commands may be entered in either upper or lower case. Those commands whose effects are potentially disastrous require the Ctrl key to be pressed in addition to a command key. The keys used were chosen to be easy to remember, wherever possible. Commands take effect immediately - there is no need to press Return and invalid commands are simply ignored. The relevant sections of the front panel display are updated after each command so any effects can be seen immediately.

MonAm is a powerful and sometimes complex program and we realise that it is unlikely that many users will use every single command. For this reason the remainder of the MonAm manual is divided into two sections - the former is an introduction to the basic commands of the program, while the latter is a full reference section. It is possible for new users and beginners to use the debugger effectively while having only read the *Overview*; but don't be intimidated by the *Reference* section.

Starting MonAm

MonAm is invoked by typing the command

```
monam [Return]
```

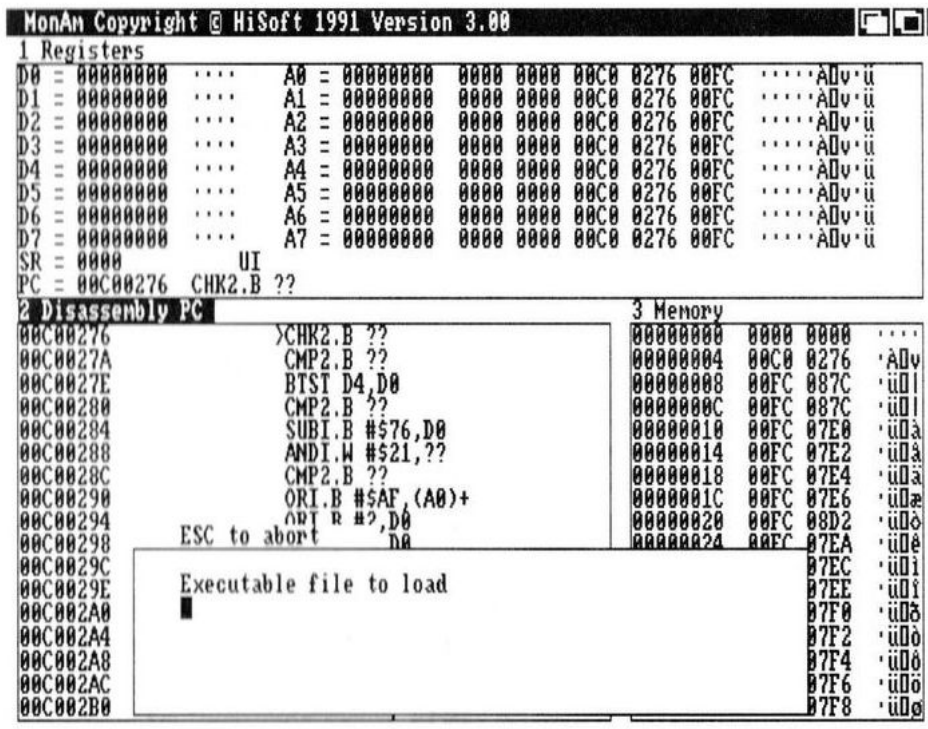
or by calling it from the HighSpeed Pascal editor.

If you start MonAm from a CLI you can include, optionally, a program name and a command line to be passed to the program. For example,

```
monam c:mytest demos/finbonacci.s [Return]
```

will cause MonAm to be invoked which will load a program called mytest and pass a filename to this program.

When MonAm has loaded, the screen will look like this:



The MonAm initial screen

If you started MonAm without asking for a program to be loaded, the prompt `Executable file to load` will appear. This gives you another chance to load a program to debug; either type the filename of the program that you want to investigate and hit Return or press Return by itself (or Esc) to quit the requester.

Should MonAm have been called from the HighSpeed Pascal editor, the program that you are developing will be loaded automatically or used from memory, if it was compiled there.

Debugging a Program

If you have asked MonAm to load a program to debug you may now be prompted for a command line, if you haven't already given one; enter the line you want or just press Return. MonAm will then try to load the program. If it fails, it will display

```
AmigaDOS error xx
```

You can use the *Load Program* command to try to load the program again.

Assuming the filename is valid, MonAm will load the executable file and any symbols within the file. After the file and its symbols have been loaded successfully, the message

```
Breakpoint
```

will appear; this is because MonAm places a breakpoint at the first instruction of the program and then executes it.

The most common command in MonAm is probably *single-step*, obtained by pressing Ctrl-Z (or Ctrl-Y if you find it more convenient, perhaps because you have a German keyboard). This will execute the instruction at the PC, shown in the Register window and, normally, also in the Disassembly window. After executing it the debugger re-displays the values of the windows, so you can watch the processor execute your program, step by step. Single-stepping is the best way of going through sections of code that are suspect and require deeper investigation, but it is also the slowest - you may only be interested in a section of code near the end of your program which could take a long tome to reach if you have to single-step all the way. There is, of course, an answer.

Breakpoints

A *breakpoint* is a special instruction placed into your program to stop it running and enter MonAm. There are many types of breakpoint but we will restrict ourselves to the simplest for now. A breakpoint may be set by pressing A B, then entering the address you wish to place the breakpoint. You can enter addresses in MonAm in hex (the default base), as a symbol, or as a complex expression. Examples of valid addresses are `1A2B0`, `prog_start`, `10+mydata` or `#12` (this means line 12 hexadecinal). If you type in an invalid address the screen will flash and allow you to correct the expression.

Having set a breakpoint you need some way of letting your program actually *run*, and Ctrl-R will do this. If will execute your program using the values of the registers displayed and starting from the PC. MonAm will be re-entered if a breakpoint has been hit, or if an exception occurs.

MonAm uses its own *screen* which is independent of your program's screen. If you use the screen gadgets at the top right of the MonAm screen you will see your current program's display. This allows you to debug programs without disturbing their output. The MonAm screen will go to the back when you run a program from within the debugger and pop to the front when a breakpoint or other exception is reached. As usual, you can drag the MonAm screen from the top to reveal your program's screen.

Windows

MonAm uses its own *windows* too, and any window may be *zoomed* to the full screen size by pressing A Z. To return to the main display press A Z again, or the Esc key. The Esc key is also the best way of getting out of anything you may have invoked by accident. The Zoom command, like all A commands, works on the *current window* which you can change by pressing Tab. You can dump the current window to your *printer* by pressing A P.

To change the *address* from which a window displays its data, press A A, then enter the new address. The locking of a window to an expression is detailed in the *Reference* section.

Quitting MonAm

To *quit* MonAm press Ctrl-C. This returns MonAm directly to the CLI. If the task you are debugging is still running or suspended when you try and quit, you will be warned. If MonAm terminates while the task under investigation is running, the machine will crash if any exception occurs subsequently. A safer way to exit is to use the Ctrl-Q command to stop the task first. If you used the Debug option from the editor then Ctrl-C will always terminate your program as well as MonAm.

We hope this overview has given you a good idea of the most common features of MonAm to let you get on with the complex process of writing and debugging assembly language programs. When you feel more confident you should try and read the following *Reference* section, probably best taken, like all medicine, in small doses.

MonAm Reference

This is the reference section to MonAm; it is a complete description of the features and commands of this powerful debugger.

Numeric Expressions

There are many occasions within MonAm when you will want to enter a numeric expression; perhaps to lock a window to an expression, to assign a value to a register or to set the start address of a window.

For these cases, MonAm has a full expression evaluator, based on that in GenAm, our assembler, including operator precedence. The main difference between it and a Pascal expression evaluator is that the default base for numbers is hexadecimal. The Pascal operators like and which have symbolic names are replaced by symbols.

The precedence table for MonAm's operators is given below:

Precedence of Operator	Operator(s)
1	monadic minus (-) and plus (+), source operators (# and ?)
2	bitwise not (~)
3	shift left (<<) and shift right (>>)
4	bitwise And (&), Or (!) and Xor (^)
5	multiply (*) and divide (/)
6	addition (+) and subtraction (-)
7	equality (=), less than (<), greater than (>), inequality (<> and !=), less than or equals (<=), greater than or equals (>=)

The comparison operators are signed and return 0 if false or -1 ($FFFFFFFF) if true. The shift operators take the left hand operand and shift it the number of bits specified in the right hand operand, vacated bits are filled with zeroes.

This precedence can be overridden by the use of parentheses (and). With operators of equal precedence, expressions are evaluated from left-to-right. Spaces in expressions (other than those within quotes - ASCII constants) are not allowed.

All expression evaluation is done using 32-bit signed-integer arithmetic, with no checking of overflow.

Numbers

Absolute numbers may be in various forms:

decimal constants, e.g. \1029

hexadecimal constants, e.g. 12f or $12f

octal constants, e.g. @730

binary constants, e.g. %1100010

character constants, e.g. 'X'

\ is used to denote decimal numbers, $ is used to denote hexadecimal numbers (the default), % for binary numbers, @ for octal numbers and single ' or double " quotes for character constants.

Character Constants

Whichever quote is used to mark the start of a string must also be used to denote its end and quotes themselves may be used in strings delimited with the same quote character by having it occur twice. Character constants can be up to 4 characters in length and evaluate to right-justified longs with null-padding if required. For example, here are some character constants and their ASCII and hex values:

Entered	Value	Hexadecimal
"Q"	Q	$00000051
'hi'	hi	$00006869
"Test"	test	$54657374
"it's"	it's	$6974277C
'it''s'	it's	$6974277C

Symbols may be referred to and are normally case-insensitive and significant to 32 characters although this can be changed with the MonAm *Preferences* command.

Registers may be referred to simply by name, such as A3 or d7 (case insensitive), but this causes a clash with certain hex numbers. To obtain such hex numbers precede them with either a leading zero or a $ sign.

There are several reserved symbols which are case insensitive, namely CODE, SP, SR, and SSP. Both A7 and SP refer to either the user- or supervisor-stack, depending on the current value of the status register. CODE refers to the first hunk in the program. This is the same as HUNK1. The second hunk is HUNK2 and so on.

Source Operators

There are two operators which allow debugging at a source code level; these are the # and ? operators.

To use these operators, you must have a source window open which is associated with the loaded executable program.

In turn, this loaded program must have been produced by a package that includes line number information in the program's HUNK.DEBUG hunk, i.e. you have used the Debugger Line numbers option. Otherwise the # and ? operators are invalid.

The # operator takes a source line number as its argument and returns the associated memory address, within the loaded program. So, say you have the source of hello.pas loaded into window 2 and the executable of hello loaded as the current program then:

```
m3=#20
```

will set the start address of window 3 to the address of line number 20 of the hello program (assuming that window 3 is not locked to another expression).

If the line number is out of range of the source (e.g. if you ask for line number 100 when there are only 90 lines of source), the result will be the address of the first or last line of the source, accordingly. If you use the # operator when there is no line number information available, the result will be 0.

The ? operator is the reverse of #; it returns the source line number, given a memory address. If the address is out of range of the code connected with the source window, ? returns a value of 0.

If you have only one source file loaded, the use of these operators is unambiguous. However, if you have loaded two or more source files into MonAm's windows, # and ? may return unpredictable results; in this case it is best to use them when one source file is open in the current window - they will then relate to this file.

These operators allow you to perform a variety of commands on a source level such as: *Set Breakpoint, Run Until* and *Lock Window*. This can make the process of debugging a complex program a far simpler and less tiresome task.

Indirection

The MonAm expression evaluator also supports indirection using the { and } symbols. Indirection may be performed on a byte, word or long basis, by following the } with a period then the required size, *which defaults to long*. If the pointer is invalid, either because the memory is unreadable or even (if word or longword indirection is used) then the expression will not be valid.

For example, the expression

```
{data_start+10}.w
```

will return the word contents of location `data_start+10`, assuming `data_start` is even. Indirection may be nested as you would nest ordinary parentheses.

Memory Registers

In addition there are 10 memories numbered `M0` through `M9`, which are treated in a similar way to registers and can be assigned to using the *Register Set* command. These are available for your own use although some have special functions as described below - you can view the memory registers by zooming the register window.

The values of memories 1 through 5 inclusive are the current start address of the relevant window (including source code displays) and assigning to them will change the start address of the display within that window. Here's a full table of the memory registers:

Memory Register	Contents
m0	the effective address referenced by the current instruction (destination where there are two)
m1	the start address of window 1
m2	the start address of window 2
m3	the start address of window 3
m4	the start address of window 4
m5	the start address of window 5
m6	spare
m7	spare
m8	the start address of any binary file that has been loaded
m9	the end address of any binary file that has been loaded

`m8` and `m9` are useful if you have loaded a binary file and then want to save it out to disk again - you do not have to remember the start and end addresses of the file, just use `m8` and `m9` when saving.

If window 1 has a register display in it, `m1` will be meaningless but will retain any previous value.

Window Types

There are five possible windows within the MonAm display and the exact contents of these windows and how they are displayed is detailed below. The allowed types of each window are:.

Window	Allowed Type(s)
1	register, memory, disassembly, source
2	memory, disassembly, source
3	memory only
4	memory, disassembly, source
5	memory only

A window can have a number of different views attached to it; you can think of the window as a *stack*, having depth.

So, in window 2, you can view a disassembly of code, a section of memory and a portion of an ASCII file, although only one of these at a time is visible. To cycle through the different views use the *Next/Previous View* commands and to create or delete a display use the *Open View* and *Close View* commands.

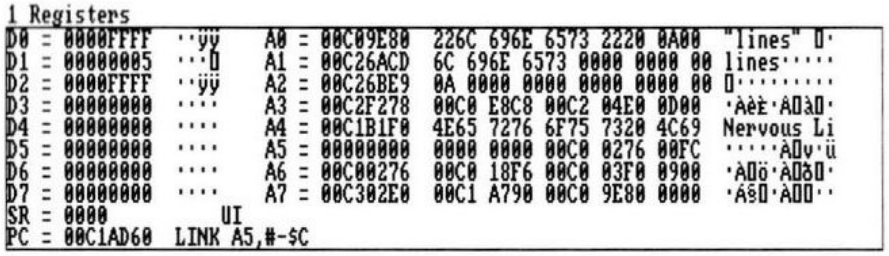
Most windows can also be *split*, either vertically or horizontally so that more, or less, can be displayed within the window - this action may hide or reveal other windows and it is best to experiment with the split commands (described below) to understand how they work.

A window can be locked to an expression so that its start address is dependent on the value of that expression - see the *Lock to Expression* command below.

You can also *zoom* a window; it will then occupy the whole of MonAm's screen.

Each type of window will now be described.

Register Window



the register window

The data registers are shown in hex, together with the ASCII display of their four bytes. The address registers are also shown in hex, together with a hex display of the memory that each register is addressing. This is word-aligned or byte-aligned as necessary, with non-readable memory displayed as **. To the right of this hex display is its ASCII interpretation.

The status register is shown in hex and in flag form, additionally with U or S denoting user- or supervisor-modes.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

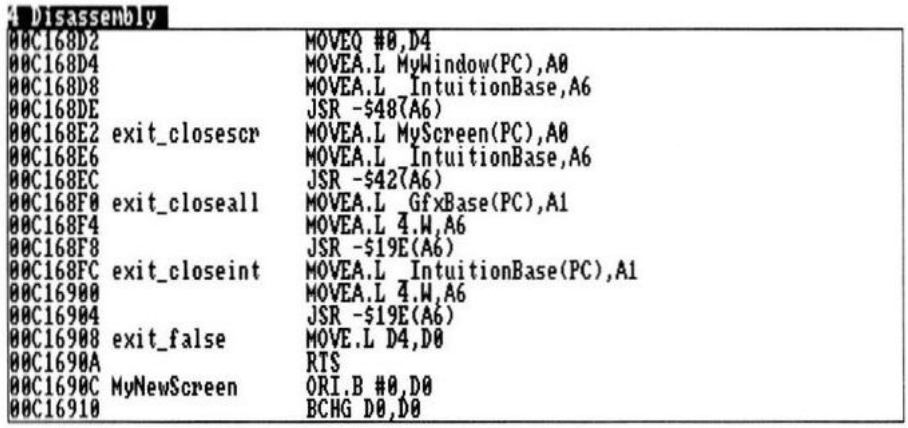
```
TST.L $12A(A3) ;00001FAE 0F01
```

signifies that the value of $12A plus register A3 is $1FAE, and that the word memory pointed to by this is $0F01. A more complex example is the display

```
MOVE.W $12A(A3),-(SP) ;00001FAE 0F01 >0002AC08 FFFF
```

The source addressing mode is as before but the destination address is $2AC08, presently containing $FFFF. Note that this display is always of a suitable size (MOVEM data being displayed as a quad-word) and when pre-decrement addressing is used this is included in the address calculations.

Disassembly Window



the disassembly window

Disassembly displays show memory as disassembled instructions to the standard described below. On the left the hex address is shown, followed by any symbol, then the disassembly itself. The current value of the PC is denoted with a >, if it is visible.

You can scroll through the disassembly window as described under *Cursor Keys* below.

If the instruction has a breakpoint placed on it this is shown using square brackets ([ ]) afterwards, the contents of which depend on the type of breakpoint.

For stop breakpoints this will be the number of times left for this instruction to execute, for conditional breakpoints it will be a ? followed by the beginning of the conditional expression, for count breakpoints it is an = sign followed by the current count and for permanent breakpoints a * is displayed.

The exact format of the disassembled op-codes is to the Motorola standard. All output is upper-case (except lower-case labels) and all numeric output is in hexadecimal, except Trap numbers. Leading zeroes are suppressed and the $ hex delimiter is not shown on numbers less than 10. Where relevant numerics are shown signed.

The only deviation from Motorola standard is the register lists shown in MOVEM instructions - in order to save display space the type of the second register in a range is abbreviated, for example

```
MOVEM.L d0-d3/a0-a2,-(sp)
```

will be disassembled as

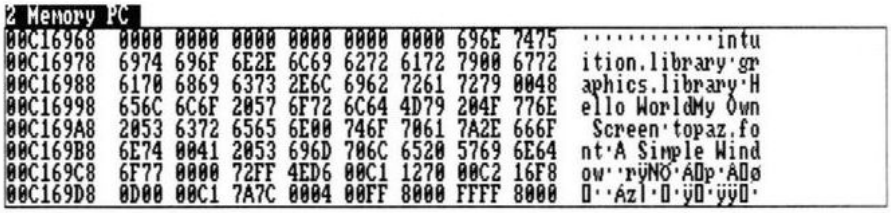
```
MOVEM.L d0-3/a0-2,-(sp)
```

Certain library calls will be shown symbolically even if no symbol information was loaded with your program. The disassembler is intelligent and recognises a MOVE into register A6 followed by a JSR using A6 so that the call, if valid, will be displayed by name; for example

```
MOVE.L 4,A6
JSR    _LVOOpenLibrary(A6)
```

The disassembler does this for the exec, graphics, dos and intuition libraries, using the special file monam.libfile. If this file is not found in the current directory or the current libs: assignment during MonAm's initialisation then such disassembly will not occur.

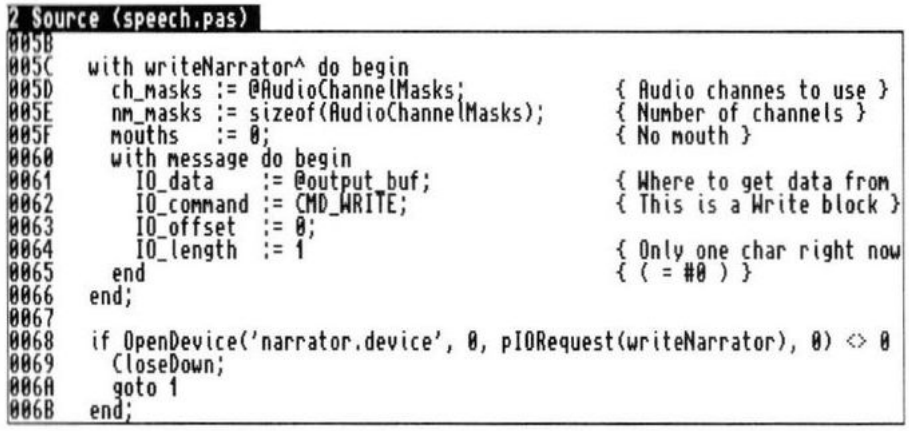
Memory Window



the memory window

Memory displays show memory in the form of a hex address, word-formatted hex display and ASCII. Unreadable memory locations are denoted by **. The number of bytes shown is calculated from the window width, up to a maximum of 16 bytes per line. You can scroll through the memory window as described under *Cursor Keys* below.

Source Window



the source window

The source window shows ASCII files in a similar way to a screen editor with the name of the file displayed in the title bar. The default tab setting is 8 though this can be toggled to 4 with the *Edit Window* command.

You can choose whether or not to display line numbers for the file and whether they are shown in decimal or hexadecimal. When line number information exists in the HUNK_DEBUG hunk of your program, you can use the medium level debugging features of MonAm to step through the source and set breakpoints within it, rather like you can with a source code debugger.

You can scroll through the source window as described under *Cursor Keys* below.

Cursor Keys

The cursor keys can be used on the current window, the action of which depends on the display type.

On a memory display all four cursor keys change the current address, by byte or line, while Shift ↑ and Shift ↓ move a page in either direction. On a disassembly display ↑ and ↓ change the start address on an instruction basis, ← and → change the address on a word basis and Shift ↑ and Shift ↓ on a page basis.

On a source-code display ↑ and ↓ change the display on a line basis, and Shift ↑ and Shift ↓ on a page basis.

Window Commands

Commands that are reached through the use of the A (right Amiga) key are normally available at any time. Many of these commands are connected with and apply to the *current window*. The current window is denoted by having an inverse title and it can be changed by pressing Tab or A plus the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

The exceptions to the above are the *Stack*, *Unstack* and *View Stack* commands which, for ease of use, are not reached through the A key and do not work on a zoomed window.

A A or M — *Set Address*

Allows you to set the starting address of a memory, disassembly or source window (the latter only if line number information exists in the HUNK_DEBUG hunk). You can use any valid expression to generate this start address e.g.

```
_main
$C227B8
StartProgram+8
PC
```

A E — *Edit View*

On a memory window this lets you edit memory in hexadecimal or ASCII. Hex editing can be accomplished using keys 1-9, A-F, together with the cursor keys. Pressing Tab switches between hex & ASCII, ASCII editing takes each keypress and writes it to memory. The cursor keys can be used to move about memory. To leave edit mode press the Esc key.

On a register display using this command is the same as using A R, *Register Set*, described shortly. Within a source window this command toggles the tab setting between 4 and 8.

You cannot edit a disassembly window.

⌘G Goto Source Line

This command works on source windows and allows you to choose the line that will appear at the top of the window. If you select a line that is beyond the end of the file, the last line will be shown at the top of the window.

⌘L Lock to Expression

This allows source (with line number information), disassembly and memory windows to be locked to a particular expression. After any exception the start address of the display is re-calculated, depending on the locked expression. Each stacked view within a window can have its own lock.

MonAm will ignore you if you try to lock a source window that refers to a program that does not have line number information attached to it.

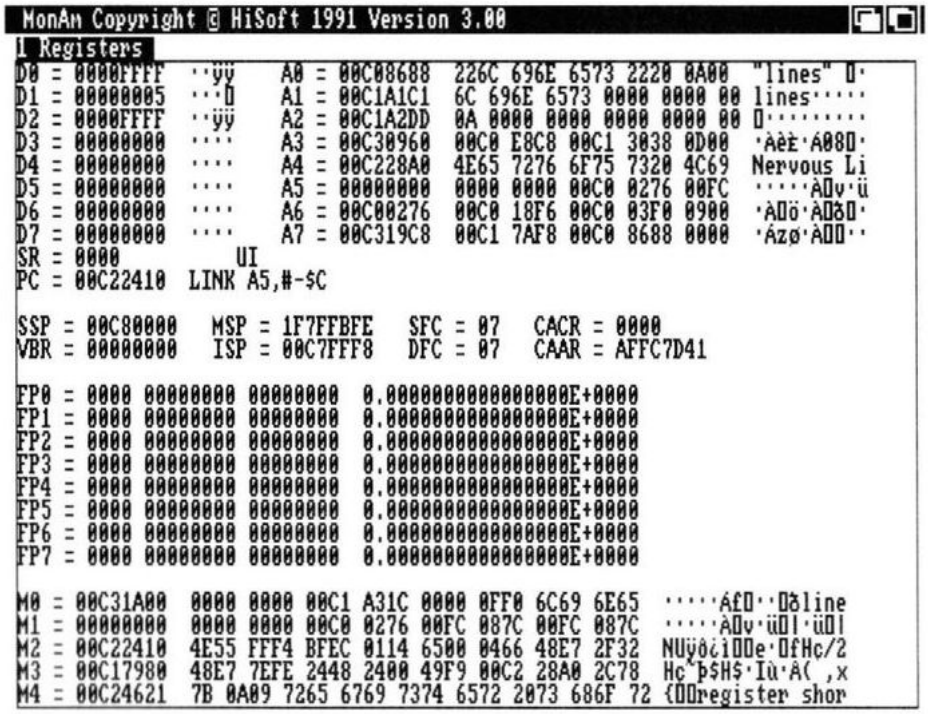
If you try to lock a source window to an expression that lies outside the address range of the source file you will be ignored. This, in fact, is very useful; it means that if you have a stack of source windows (see below for details of stacking windows) which make up the executable that you are debugging and you lock each display to the `PC`, you will be able to trace the path of the program through each source file.

If an instruction in the top view calls a subroutine in the stack, the top view will not change but, if you then view the relevant stacked view, it will change to show you the called subroutine.

To unlock, simply enter a blank string.

You can lock one window to another window by using the memory registers such as `M2`. You can even lock a window to the indirection of its own memory register (e.g. `{m2}`) which might be useful to step through a linked list (in conjunction with the Esc key to update the window each time).

Zooming a register window shows some extra information (which depends on the processor type) and the memory registers (m0 - m9):



the zoomed register display (on a 68020 machine)

A zoomed window behaves differently from a normal window in that, as you scroll through it, it does not update the associated memory register (m1 to m5). Also, if you change the value of the memory register while in a zoomed window , the start address of the display will not change. Think of a zoomed window as only temporary.

Shift-. *Open View*

Creates a new view on the current window and numbers it accordingly. The type of the new view will be the same as the previous one if this is possible.

The display will be numbered xa, xb, xc, xd etc. where x is the number of the window e.g. if you stack a new display on window 2, it will be numbered 2b with the original display being numbered 2a. Remember, though, that there is only one memory register per window, but you can lock each display to a different expression. This gives a tremendous amount of flexibility.

This command does not work on a zoomed window.

The associated memory register is bound to the top view only, although all locks on all views are re-calculated where necessary.

Shift-, *Close View*

Removes the visible display from the current window's display list, unless there is only one display attached to this window, in which case the command does nothing. If you close a view on a source window, the source file will be un-loaded from memory and a disassembly window will replace the closed source view.

All other displays attached to this window will be re-numbered if necessary i.e. if you remove display 2c from (2a, 2b, 2c, 2d), 2d will be re-numbered to be 2c.

This command does not work on a zoomed window.

. and , *Next/Previous View*

These two commands allow you to cycle through views that have been stacked onto a window. Pressing . (full stop or period) cycles forward through the available displays whilst , (comma) cycles backwards. Both will roll round in a loop.

For example, say you have 3 displays stacked on window 4 (4a Source, 4b Memory and 4c Disassembly) and you are currently displaying 4b Memory. Press . and 4c Disassembly will appear, press . again and you will see 4a Source.

These commands do not work on a zoomed window.

Pressing Esc will update all the window displays, if necessary and re-calculate the addresses to which any windows and views are locked.

This can be very useful in many cases; for example say you have window 3 locked to {m5} (the address pointed to by window 5) and you then scroll through window 5. Normally this will not update window 3. However, all you have to do is to press Esc when you want to update window 3 (and all the other windows).

You can press the Esc key while your program is running to see how the machine state is changing (assuming that the MonAm screen is at the front).

Other A Commands

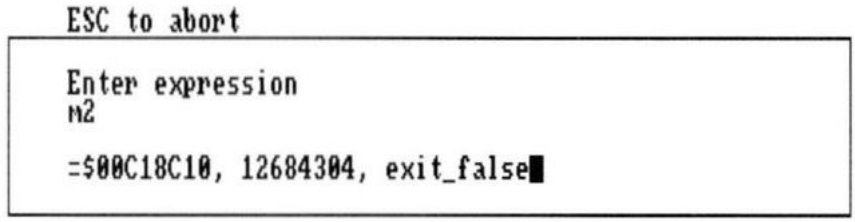
All A (right Amiga) commands (like the window commands described above) are available for use at any time whilst you are using MonAm. There are a few other such commands that are not related to the current window:

AB — *Set Breakpoint*

Allows the setting of any type of breakpoint, described later under *Breakpoints*.

A0 or 0 — *Show Other Bases*

This prompts for an expression and displays its value in hexadecimal, decimal and as a symbol if relevant.



example of Show Other

Allows any register to be set to a value, by specifying the register, an equals sign and its new value. It can also be used to set the value of the memory registers. For example the line

```
a3=a2+4
```

sets register A3 to be A2 plus 4 whereas:

```
m3=m2
```

will set the value of the window 3 register to be the same as the window 2 register. All windows will then be re-drawn, which may cause a display that you did not expect if, say, the display in window 3 is locked to an expression.

You can also use this to set the start address of a window when in zoom mode so that, on exit from zoom mode, the relevant window starts at the required address.

Screen Switching

MonAm uses its own Screen display and will always make itself the front and active window whenever an exception (including breakpoints) occurs.

V View other Screen

This will put the MonAm screen to the back, normally showing your program's screen. Pressing any key will return the MonAm screen (so long as you have not activated any other window).

Breakpoints

Breakpoints allow you to stop the execution of your program at specified points within it. MonAm allows up to eight simultaneous breakpoints, each of which may be one of five types. When a breakpoint is hit MonAm is entered and it then decides whether to halt execution of your program (when it will enter the front panel display) or continue; this decision is based on the type of the breakpoint and the state of your program's variables.

Simple Breakpoint [1]

These are one-off breakpoints which, when executed, are cleared and cause MonAm to be entered.

Stop Breakpoint [n]

These are breakpoints that cause program execution to stop after the break-pointed instruction has been executed a specified number of times. In fact a simple breakpoint is really a stop breakpoint with a count of one.

Count Breakpoint [=]

Merely counters; each time such a breakpoint is reached a counter associated with it is incremented, and the program will resume. These breakpoints are more like monitors - they never cause a program to stop and are useful for profiling.

Permanent Breakpoint [*]

These are similar to simple breakpoints except that they are never cleared - every time execution reaches a permanent breakpoint MonAm will be entered.

Conditional Breakpoint [?]

The most powerful type of breakpoint; these allow program execution to stop at a particular address only if an arbitrarily complex set of conditions apply. Each conditional breakpoint has associated with it an expression (conforming to the rules already described). Every time the breakpoint is reached this expression is evaluated, and if it is non-zero (i.e. true) then the program will be stopped, otherwise the program will continue.

⌘B — Set Breakpoint

This is a window command allowing the setting or clearing of breakpoints at any time. The line entered should be one of the following forms, depending on the type of breakpoint required:

`<address>`

will set a simple breakpoint.

`<address>,<expression>`

will set a stop breakpoint at the given address, which will execute `<expression>` times. The expression is evaluated before the program is executed.

`<address>,=`

will set a count breakpoint. The initial value of the count will be zero.

`<address>,*`

will set a permanent breakpoint.

`<address>,?<expression>`

will set a conditional breakpoint, using the given expression.

`<address>,-`

will clear any breakpoint at the given address.

Breakpoints cannot be set on addresses which are odd, unreadable, or within ROM.

Every time a breakpoint is reached, regardless of whether the program is interrupted or resumed, the program state is remembered in the History buffer, described below

Help — *Show Help and Breakpoints*

This displays the current breakpoints, task status, its segment list (showing where your program is), free memory and the system memory list.

UK A500 1.2 users (who cannot use the Help key) can also obtain this command by pressing A H.

Ctrl-B — *Simple Breakpoint*

Included mainly for compatibility with MonAm 1, this sets a simple breakpoint at the start address of the current window, so long as it contains a disassembly display. If a breakpoint is already there then it will be cleared.

U — *Run Until*

This prompts for an address and a breakpoint specifier (1, n, =, *, or ?). The chosen type of breakpoint is then placed at the given address and program execution resumed.

Ctrl-K — *Kill Breakpoints*

Clears all set breakpoints. This is also done automatically when you quit MonAm with a task still running.

A command that places a simple breakpoint at the instruction *after* the instruction at the PC and resumes execution from the PC. This is particularly useful for DBF-type loops if you don't want to go through the loop, but just want to see the result after the loop is finished.

Ctrl-X Stop executing

This is a command to stop your task while it is executing. It does this by forcing the Trace bit to be set, so you will get a Trace exception. While this does work, be very careful if you stop a task in the middle of some AmigaDOS ROM routines, particularly signal handling and message passing.

The above command accesses memory fields that are not guaranteed to remain the same for different versions of the Amiga® operating system.

This command was Ctrl-S in MonAm version 1.

History

MonAm has a *history buffer* in which the machine status is remembered for later investigation.

The most common way of entering data into the history buffer is when you single-step, but in addition every breakpoint reached and every exception caused enters the machine state into the buffer. The various forms of the Run command also cause entries to be made into this buffer.

The history buffer has room for five entries - when it fills the oldest entry is removed to make room for the newest entry.

This opens a large window displaying the contents of the history buffer. All register values are shown including the PC as well as a disassembly of the next instruction to be executed.

If a disassembly in the History display includes an instruction which has a breakpoint placed on it, the [ ]s will show the *current* values for that breakpoint, not the values at the time of the entry into the history buffer.

Quitting MonAm

Ctrl-C or A Q — Exit MonAm

This exits MonAm, returning control to whatever task invoked MonAm. All breakpoints are killed although, if the task you are debugging is still running or suspended when you try and quit, you will be warned. If MonAm terminates while the task under investigation is running, the machine will crash should any exception occurs subsequently. A safer way is to use the Ctrl-Q command to stop the task first.

If the Debug option has been used from the HighSpeed Pascal editor then MonAm will terminate automatically when the program it is debugging has terminated.

Ctrl-Q — Quit a program

This is a way of forcing the task being debugged to quit. This can be hazardous to use, and should only be done as a last resort. If your program is terminated in this way it will not clean up, and thus not de-allocate any memory it was using or close windows etc.

The above command accesses memory fields that are not guaranteed to remain the same for different versions of the Amiga® operating system.

Ctrl-L — Load Program

This will prompt for a filename and a command line and will attempt to load the file ready for execution. If MonAm has already loaded a program it is not possible to load another until the former has terminated.

The file to be loaded must be an executable file. Use the *Load Binary File* command if you wish to edit other file types.

This command is not available if MonAm has been invoked using Debug from the editor.

B — Load Binary File

This will prompt for a filename and an optional load address (separated by a comma) and will then load the file where specified. If no load address is given then memory will be allocated from the system. M8 will be set to the start address of the loaded file and M9 to the end address.

This is a change from previous versions of MonAm, where M0 and M1 were set to the start and end addresses of the loaded file.

S — Save Binary File

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the *Load Binary File* command

```
<filename>,M8,M9
```

may be specified, assuming of course that M0 and M1 have not been re-assigned.

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within MonAm. This can be loaded into window 2 or window 4. If the loaded program has line number information relevant to this source file, you will be able to use line number operators on this display to step through the source code, set breakpoints within it etc.

A new view on this window will be opened if the window already contains an ASCII file, otherwise the text will replace the current window. You can unload a source window using the *Close View* command.

The source window will be locked automatically to the PC.

Memory for source code displays is taken from the system so sufficient free memory must be available.

Executing Programs

Ctrl-R *Return to program / Run*

This runs the current program with the given register values at full speed and is the normal way to resume execution after entry via a breakpoint or an exception.

Ctrl-Z *Single-Step*

Single-steps the instruction at the PC with the current register values. Single-stepping a Trap, Line-A or Line-F opcode will, by default, be treated as a single instruction.

Ctrl-Y *Single-Step*

Identical to Ctrl-Z above but included for the convenience of users of German keyboards.

This interprets the instruction at the PC using the displayed register values. It is similar to Ctrl-Z but obeys BSRs, JSRs, Traps, Line-A and Line-F calls as if one instruction, re-entering the debugger on return from them to save stepping all the way through the routine or trap. It works on instructions in ROM or RAM.

Ctrl-S *Skip Instruction*

Ctrl-S increments the PC register by the size of the current instruction thus causing it to be skipped. Use this instead of Ctrl-Z when you know that this instruction is going to do something it shouldn't.

R *Run (various)*

This is a general *Run* command and prompts for the type of execution, selected by pressing a key.

Run G Go

This is identical to Ctrl-R and resumes the program at full speed.

Run I Instruction

This executes the entered number of instructions remembering information in the History buffer and then returning to MonAm.

Traps are treated as single-instructions.

Searching Memory

G *search memory (Get a sequence)*

You will see the prompt Search for B/W/L/T/I?, standing for Bytes, Words, Longs, Text and Instructions.

If you select B, W or L you will then be prompted to enter the sequence of numbers you wish to search for, each separated by commas. MonAm is not fussy about word-alignment when searching, so it can find longs on odd boundaries, for example.

If you select T you may search for any given text string, for which you will be prompted. The search will be case-dependent or case-independent, as you have chosen.

If you select I you can search for part or all of the mnemonic of an instruction, for example if you searched for $14(A you would find an instruction like MOVE.L D2,$14(A0). The case of the string you enter *is* un-important unless you have chosen it to be so, but you should bear in mind the format that the disassembler produces, e.g. always use hex numbers, refer to A7 rather than SP and so on.

Having selected the search type and parameters, the search begins, control passing to the *Next* command, described below.

Searching Source-Code Windows

If the G command is used on a source-code window the T sub-command is automatically chosen and if the text is found the window will re-display the line containing it.

N — *find Next*

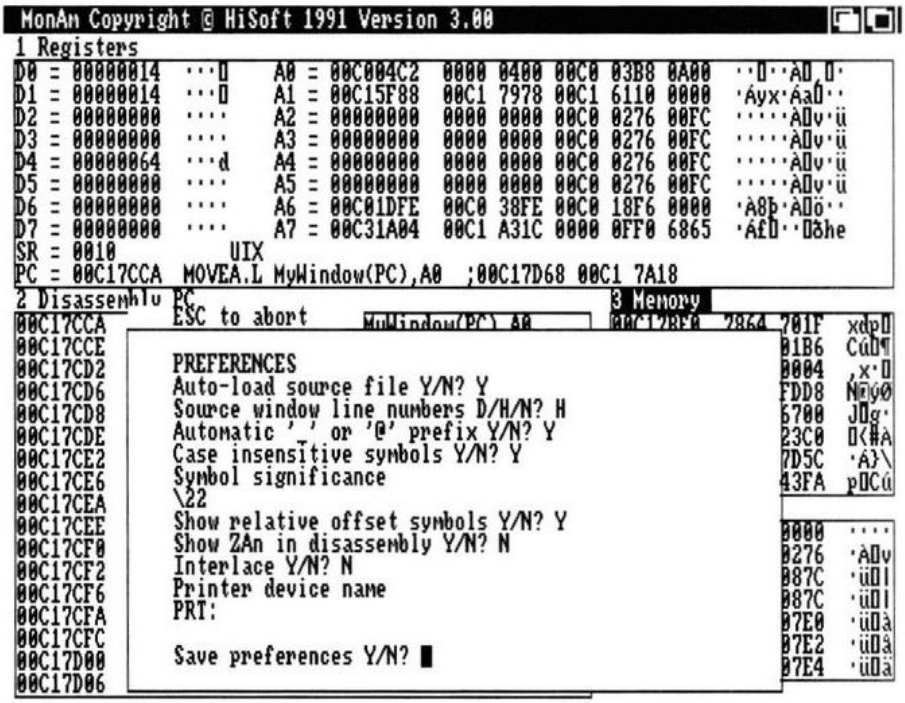
N can be used after the G command to find subsequent occurrences of the search data. With the B, W, L and T options you will always find at least one occurrence, which will be in the buffer within MonAm that is used for storing the sequence. With the T option you may also find a copy in the system keyboard buffer. With these options, the Esc key is tested every 64k bytes and can be used to stop the search. With the I option, which is very much slower, the Esc key is tested every 2 bytes.

The search will start just past the start address of the current window (except register windows) and if an occurrence is found re-display the window at the given address.

The search area of memory goes from 0 to the end of chip memory, then $F80000 to $FFFFFF (the ROM) then any additional RAM.

Ctrl-P Preferences

This permits control over various options within MonAm. The first three require Y/N answers, pressing Esc aborts or Return leaves them alone.



the Preferences display

Auto-load source file

When switched to Yes, upon loading a program, MonAm will attempt to load the first source file associated with the program. This will only occur if the executable file contains line number debugging information and the source file can be found in the current directory. The new source file window will then be locked to the Program Counter in order to track program flow. This is of most use when debugging a program generated from a single source file.

Source window line numbers

Affects whether line numbers are shown for all debugger source windows. You may select No line numbers, Decimal numbers or Hex numbers. Hexadecimal is often the preferred setting because by default, MonAm treats all numbers as hex. Decimal line numbers, used with the # operator for example, require a prefix of backslash.

Automatic '_' or '@' prefix

This is provided mainly for the convenience of C compiler users. With it enabled, MonAm will automatically add a leading underscore or @ character to the appropriate symbols. However, symbols without the leading character will still take precedence.

Case insensitive symbols

MonAm version 3 defaults to using *case insensitive* symbols, i.e. upper and lower case characters are not distinguished between. Selecting No will mean that you must match the case of each symbol character exactly as with previous versions of MonAm.

Symbol significance

This prompts for the significant length of symbols, which is normally 32 but may be reduced to as low as 8 or increased if required. Although reducing this can save some typing, using too low a value can make some symbols impossible to select.

Show relative offset symbols

This option defaults to Yes and affects the disassembly of *address register indirect with offset* addressing modes, i.e. `xxx(An)`. With the option on, the current value of the given address register is added to the offset then searched for in the symbol table. If found it is disassembled as `symbol(An)`. This option is very useful for certain styles of assembly language programming as well as high level languages which use a base register as a major offset, such as SAS/C which uses A4 as a pointer to the merged data section.

Show ZAn in disassembly

Is normally switched off but advanced programmers may wish to enable the display of the normally hidden Z registers used by some 680x0 instructions.

Interlace

Allows you to select a double height interlaced screen for MonAm. This option will take effect after preferences have been saved and MonAm restarted. MonAm will normally replicate the Workbench screen's format.

Printer device name

This lets you set the device that MonAm uses for its printer commands. The default is `PRT:`, the system printer device configured through Preferences. You may specify an AmigaDOS filename in order to re-direct printing to disk.

Save preferences

Reply `Y` to this command to save your current preferences to the file MonAm.prefs in the current directory. When MonAm loads it will read your preferences from this file. MonAm.prefs is firstly searched for in the current directory, then in the ENV:Devpac directory, in a similar way to the editor preferences file. Note that the ENV:Devpac directory is used rather than ENV:Pascal. This is for the benefit of users of our Devpac assembly language development system; there is only one debugger prefrences file and the debuggerd from either package may be used interchangeably.

I *Intelligent Copy*

This copies a block of memory to another area. The addresses should be entered in the form

```
<start>,<inclusive_end>,<destination>
```

The copy is intelligent in that the block of memory may be copied to a location which overlaps its previous location.

No checks at all are made on the validity of the move; copying to non-existent areas of memory is likely to crash MonAm and corrupting system areas may well crash the machine.

L — *List Labels*

This opens up a large window and displays all loaded symbols. Any key displays the next page, pressing Esc aborts. The symbols will be displayed in the order they were found on the disk (or in memory if using the Debug option from the editor).

Ctrl-U name — *Unload symbols*

This command can only be used if you are debugging a task which had a symbol table loaded with it. What it does is de-allocate the memory used for storing the symbols, freeing it for the system to use. This can be very useful if memory is tight while debugging a larger program, as you can load it, together with symbols, set a breakpoint at a symbolic address, then lose the labels before letting it run. Of course once you hit your breakpoint you won't have any symbols.

W — *Fill Memory With*

This fills a section of memory with a particular byte. The range should be entered in the form

```
<start>,<inclusive_end>,<fillbyte>
```

The warning described previously about no checks applies equally to this command.

P — *Disassemble to Printer/Disk*

This command allows the disassembly of an area of memory to printer or disk, complete with original labels and, optionally, an automatic list of labels created by MonAm, based on cross-references. The first line should be entered as

```
<start_address>,<end_address>
```

The next line prompts for the area of memory used to build the cross-reference list, which should be left blank if no automatic labels are required else should be of the form

```
<buffer_start>,<buffer_end>
```

Next is the prompt for data areas which will be disassembled as DC instructions, of the form

```
<data_start>,<data_end>[,<size>]
```

The optional size field should be B, W or L, defaulting to L, determining the size of the data. When all data areas have been defined, a blank line should be entered.

Finally a filename prompt will appear; if this is blank all output will be to the printer, else it will be assumed to be a disk file.

If automatic labels were specified there may be a delay at this point while the table is generated. Automatic labels are of the form Lxxxxx where xxxxx is the actual hex address.

Printer Output

This is of the form of an 8 digit hex number, then up to 10 words of hex data, 12 characters of any symbol, then the disassembly itself. Printer output may be aborted by pressing Esc.

Disk Output

This is in a form directly loadable by GenAm, the assembler supplied with Devpac, consisting of any symbol, a tab, then the disassembly itself, with a tab separating any operand from the op-code. If you are disassembling an area of memory without loaded symbols then the XREF option should be used else no symbols will appear at all in the output file. Pressing Esc or a disk error will abort the disassembly.

M — *Modify Address*

Included for compatibility with MonAm 1, equivalent to A.

O — *Show Other Bases*

Included for compatibility with MonAm 1, equivalent to 0.

D — *Change Drive & Directory*

This allows the current drive and sub-directory to be changed.

Command Summary

Window Commands

⌥ A	Set Address
⌥ B	Set Breakpoint
⌥ E	Edit View
⌥ G	Goto Source Line
⌥ L	Lock to Expression
⌥ P	Print Window
⌥ R	Register Set
⌥ S	Split Window
⌥ T	Change Type
⌥ W	Widen Window
⌥ Z	Zoom Window
Shift-.	Open View
Shift-,	Close View
. and ,	Next/Previous View
Esc	Update all Windows

Breakpoints

Ctrl-A	Breakpoint After
Ctrl-B	Simple Breakpoint
Ctrl-K	Kill Breakpoints
Ctrl-X	Stop Executing
⌥ B	Set Breakpoint
U	Run Until
Help	Show Help and Breakpoints

Loading and Saving

Ctrl-L	Load Program
A	Load ASCII File
B	Load Binary File
S	Save Binary File

Executing Programs

Ctrl-R	Return to program / Run
Ctrl-S	Skip Instruction
Ctrl-T	Trace Instruction
Ctrl-Y	Single-Step
Ctrl-Z	Single-Step
R	Run (various)

Searching Memory

G	Search Memory (Get a sequence)
N	Find Next

Miscellaneous

Ctrl-C or Q	Exit MonAm
Ctrl-Q	Quit a program
Ctrl-P	Preferences
Ctrl-U	Unload symbols
0 or 0	Show Other Bases
D	Change Drive & Directory
H	Show History Buffer
I	Intelligent Copy
L	List Labels
M	Modify Address
P	Disassemble to Printer/Disk
V	View other Screen
W	Fill Memory With

Bug Hunting

There are probably as many strategies for finding bugs as there are programmers; there is really no substitute for learning the hard way, by experience. However, here are some hints which we have learnt, the hard way!

Firstly, a very good way of finding bugs is to look at the source code and think. The disadvantage of reaching first for the debugger, then second for the source code, is that it gets you into bad habits. You may switch to a machine or programming environment that does not offer low-level debugging, or at least not one as powerful you are used to.

If a program fails in a very detectable way, such as causing an exception, debugging is normally easier than if, say, a program sometimes doesn't quite work exactly as it should.

Many bugs are caused by a particular memory location being stepped on. Where the offending memory location is detectable, by producing a bus error, for example, a conditional breakpoint placed at one or more main subroutines can help greatly. For example, suppose the global variable at address $123456 is somehow becoming odd during execution, the conditional expression could be set up as

```
{$123456}&1
```

Count breakpoints are a good way of tracking down bugs *before* they occur. For example, suppose a particular subroutine is known to eventually fail but you cannot see why, then you should set a count breakpoint on it, then let the program run. At the point where the program stops, because of an exception say, look at the value of the count breakpoint (using Help). Terminate the program, re-load it, then set a stop breakpoint on the subroutine for that particular value or one before it. Let it run, then you can follow through the sub-routine on the very call that is fails on, to try and work out why.

Good luck!

Exceptions

MonAm employs the 680x0 processor *exceptions* to stop runaway programs and to single-step, so at this point it would be useful to explain them and detail what normally happens when they occur on an Amiga.

While using the 680x0 processors, there are various types of exception that can occur, some deliberately, others accidentally. An exception is a special condition that takes priority over normal processing - it might be an interrupt from an external device, an illegal instruction, an address error, a co-processor violation or a number of other pre-defined events.

When an exception occurs the processor's context is saved on the supervisor stack and execution is then transferred to any one of 256 different addresses, held in the *exception table* (on the 68010 upwards, the address of the start of this table is held in the *vector base register*, or *VBR*). This table is set up by the Amiga's operating system so that an exception effectively transfers control to Exec, which is part of the Amiga's operating system.

The operating system then looks to see if the task that was running when the exception occurred has installed an *exception handler* i.e. the task wants to handle exceptions itself. If it has, control is passed to that exception handler; this is how MonAm traps exceptions because MonAm has attached such an exception table to the task that it has executed.

Unfortunately, there a few exceptions that MonAm cannot trap because Exec does not pass them on - in these cases the operating system does what it normally does in the absence of an exception handler, it produces a `Software Error` alert (the dreaded Guru).

MonAm actually uses two of the exception vectors itself, one to set breakpoints in programs and the other to allow single-stepping.

The various forms of exceptions, their usual results, and what happens when they occur with MonAm active is shown in the following table, which is a summary of the exception table. Note that the first 64 vectors are defined by Motorola.

Exception no.	Exception	MonAm active
0	reset initial interrupt stack pointer	n.a.
1	reset initial program counter	n.a.
2	bus error	trapped
3	address error	trapped
4	illegal instruction	breakpoint
5	zero divide	trapped
6	CHK instruction	trapped
7	TRAPV instruction	trapped
8	privilege violation	trapped
9	trace	single-step
10	line 1010 emulator	trapped
11	line 1111 emulator	trapped
12	reserved	trapped
13	co-processor protocol violation	trapped
14	format error	guru
16-23	reserved	trapped
24	spurious interrupt	guru
25-31	level x interrupt autovector, where x=26-exception no.	not trapped
32-47	trap #0 to #15	not trapped
48	FPCP branch or set on un-ordered condition	trapped
49	FPCP inexact result	trapped
50	FPCP divide by zero	trapped
51	FPCP underflow	trapped
52	FPCP operand error	trapped

Exception no.	Exception	MonAm active
53	FPCP overflow	trapped
54	FPCP signalling NAN	trapped
55	reserved	trapped
56	MMU configuration error	guru
57	68851 illegal operation	guru
58	68851 access level violation	trapped
59-63	reserved	trapped
64-255	user defined vectors	unused

Restrictions

As it runs as a process MonAm relies on the exec, intuition and graphics libraries. If your program starts destroying memory to which it has no right it is possible for it to corrupt fatally something the system needs so that when MonAm is entered after an exception the machine will crash. Fortunately this type of error is rare, usually address errors occur before programs start destroying memory.

Owing to a hardware feature, a word- or longword-access on odd memory locations 1 to 7 inclusive will cause a complete machine crash. There seems to be nothing we can do to prevent this.

Environment

When a program is invoked from MonAm it is set up to look as if it has been run from the CLI, not the Workbench.

Don't try and run the standard system programs from within MonAm, such as `dir`. These programs rely on undocumented registers (particularly A5) and memory areas which MonAm cannot emulate.

Single stepping

MonAm cannot single-step or breakpoint any code when executing in Supervisor mode. This is because the exec exception handler checks for an exception in supervisor mode, and will put up a Guru alert if this is the case. If not it will enter MonAm and work normally.

If your program creates another program or task you cannot use MonAm to breakpoint it or single-step. MonAm can only debug the program that was specified when it initially loaded.

Taking over

MonAm is an operating system debugger in that it relies upon the system for its display and input etc. If a program tries to shut down the system or take over resources (such as the screen display) for its own exclusive use, MonAm will no longer function correctly.

An example of this is a games program which takes over the display. In some cases it is possible to work round this by using the correct operating system conventions for accessing shared resources (for testing only, if desired) such as opening an Intuition screen for the display.

Chapter 6
Other Tools

Libmaker

Libmaker lets you make a customised Pascal.Lib file that the compiler and integrated environment can load to give you fast compilations that do not need to load the .unit files from disk. The system unit and printer unit are already built into the compiler; these are not included in the Pascal.lib file.

You will probably want your units drawer as the current directory before you run Libmaker.

The command line syntax is as follows:

```
Libmaker {Filename | "WITH" Filename} LibFilename
```

The `filenames` are the names of the units that are to be included in the library; if they do not contain a period (`.`) then `.unit` will be appended. `LibFileName` is the new library that is to be created. Prefixing a filename with `WITH` causes it to be treated as a list of filenames rather than a file itself; this is generally the easiest way of using LibMaker particularly if you are including the operating system units. We have supplied several example `WITH` files to help you.

If you are using the standalone compiler you need to call the destination library, `Pascal.lib`, whereas if you are using the integrated environment you can use any filename you wish, so long as you remember to alter the appropriate item in the Editor Settings requester.

Examples

```
Libmaker with AllUnits Pascal.lib
```

makes the entire library as supplied.

```
Libmaker myunit pascal.lib
```

makes the minimum library with the addition of myunit.unit.

```
Libmaker system system2 myunit pascal.lib
```

makes a library with system.unit, system2.unit and myunit.unit.

```
libmaker with AllUnits myunit1 myunit2.unit mypascal.lib
```

makes the complete library, calling it mypascal.lib, with the addition of myunit1 and myunit2.

FD to Pascal converter

Function Description files are the standard method of detailing function calls, procedures and parameter passing for an Amiga® library, device or resource. They are supplied by Commodore for the operating system and by software houses for third-party extension libraries.

To allow you to call Amiga® libraries other than the operating system ones directly supported by HighSpeed Pascal, we include an 'FD to Pascal' conversion utility. What this does is to generate a skeleton Pascal unit which allows you to call all of the library functions. In order to use it you must first obtain an FD file for the relevant library, e.g. `midi_lib.fd`.

Note that this conversion does not generate any structures or parameter and return types since this information does not exist within the FD file. Function parameters use only the generic types `LongInt` and `Pointer` and all functions return a `LongInt` (procedures are never generated). Appropriate types must be added manually after conversion.

As with the Amiga® units your program must first open the library to obtain a 'base pointer'. This *must* be assigned to the base pointer variable defined within the unit before calling any functions or the machine will crash.

Usage

The command line syntax accepted by FDToPascal is simply:

```
FDToPascal <input_lib.fd> [<output.pas>]
```

By default, the generated Pascal file has the same name as the FD file (with the `_lib.fd` extension removed and `.pas` added) although you may wish to over-ride this with the optional output name parameter.

For example, to convert the req.library FD file you might use:

```
FDToPascal req_lib.fd
```

or, if you wish to add directory specifications:

```
FDToPascal DF0:fd/req_lib.fd :units/Req.pas
```

Unit format

The generated unit includes an 'interface' section describing each function and its parameters, followed by an 'implementation' section with inline assembler 'stubs' which place parameters in the correct registers and call the library via a base pointer. The base pointer name used is taken from the `##base` directive in the FD file. This must be present or an error will be reported. No entries are generated for `##private` functions as these are (by definition) not for application use.

For example, the Intuition library function `ItemAddress` appears in the FD file as

```
ItemAddress(menuStrip,menuNumber)(a0,d0)
```

which is converted to

```
Function ItemAddress(menuStrip: Pointer,
          menuNumber: LongInt): LongInt;
XASSEMBLER;
ASM   move.l      a6,-(sp)
      movem.l     $8(sp),d0/a0
      move.l      IntuitionBase,a6
      jsr         -$90(a6)
      move.l      d0,$10(sp)
      move.l      (sp)+,a6
END;
```

More complex parameter lists will may be optimised in various ways. Please note that the code generated may differ slightly with future versions of the utility.

Using other libraries

In order to make full use of a library, there are two main areas of modification for units generated by FDtoPascal.

Firstly, structure and type definitions should be added. These can be translated from high level language header files supplied with the library such as C or Modula-2. For details about C declarations, please refer to the *C for Pascal Programmers* section in the next chapter.

The second problem is that of modifying the stubs to add type information and converting functions to procedures where necessary. To demonstrate this, here is the above Intuition stub with the correct type information added.

```
Function ItemAddress(menuStrip: pMenu,
          menuNumber: Integer): pMenuItem;
XASSEMBLER;
ASM   move.l     a6,-(sp)
      lea  8(sp),a6
      move.w     (a6)+,d0
      move.l     (a6),a0
      move.l     IntuitionBase,a6
      jsr   -$90(a6)
      move.l     d0,$E(sp)
      move.l     (sp)+,a6
END;
```

The most obvious change is in the parameter types. However, this necessitates further modifications. Firstly, because an integer is passed as 16 bits rather than a 32 bit longword, a move multiple cannot be used. Secondly, the change in stack usage that this causes gives a different offset for the return value.

For library calls documented as having no return value, you should change the word `Function` to `Procedure` and remove the return type. In addition to this, the `move.l d0` line towards the end of the stub must be removed.

Chapter 7
Operating System Support

The Amiga Units

HighSpeed Pascal is supplied with a number of units specific to the Amiga which give access to the machine's operating system. Whilst it is outside the scope of this manual to give a complete description of the system software, this section does give an overview of the facilities available and within which subsystem and Pascal unit you can find them.

Many of the example programs supplied on your HighSpeed Pascal disks make use of these units and as such, serve as an excellent starting point for writing your own Amiga programs. For a complete guide to system you should refer to the Amiga ROM Kernel Manuals or one of the many books available on programming the Amiga (see the *Bibliography*).

The operating system itself consists of a hierarchical system of disk loaded and ROM resident function libraries, device drivers and shared hardware resources all working under the management of the system executive, responsible for the multitasking and message passing facilities of the machine.

In order for your program to use the facilities of a given library, device or resource you must first 'open' the relevant subsystem. When your program starts up, the Exec and DOS libraries have been opened for you and ready for use. You can then use the `OpenLibrary`, `OpenDevice` and `OpenResource` functions of Exec (accessed from Pascal through the `Exec` unit) to obtain access to the other parts of the system software.

Each of the supplied units encapsulates a single library, device or resource including all of its functions, procedures, data structures and constants. These units are Pascal equivalents of the C language header files as supplied by Commodore and documented in the ROM Kernel Manuals. The corresponding header files for each unit are listed alongside the unit description for reference purposes.

In order to fully understand the system documentation and header files, it is most helpful to have some knowledge of the C language and we will be referring to C throughout the rest of this chapter - there is even a complete section on C for Pascal programmers towards the end of this chapter.

Conventions

In writing the HighSpeed Pascal Amiga operating system units we have been most careful to preserve the information contained in the Commodore system include files. Where possible, all types, constants and functions use identical naming and syntax to their C counterparts. The conventions used throughout the Amiga units are detailed below.

Types

The system headers make extensive use of a few basic types including `LONG`, `WORD` and `BYTE` along with their unsigned equivalents, `APTR` etc. However, in order to make the units more accessible to Pascal programs, the appropriate HighSpeed Pascal types have been substituted although these types still exist and you may use them within your own programs.

All record types use a prefix of `t`, for example `tRastPort` is the type name of the `RastPort` structure used by the Graphics unit.

Corresponding pointer types also exist with a prefix of `p`, e.g. `pRastPort` is defined as a pointer to type `tRastPort`. All records and functions which use such a pointer will take a `pRastPort` thus preventing incorrect type errors. The system units only declare the pointer types which they themselves use. You may wish to add similar types to your own programs for other pointers such as

```
Type ptimerequest = ^ttimerequest;
```

Strings

In general, the Amiga uses 'null-terminated' strings as used by the C language, i.e. any number of characters followed by a terminating zero byte. For comparison, Pascal utilises a single length byte followed by that number of characters. One notable exception to this is AmigaDOS's use of BCPL strings which are identical to Pascal strings apart from the important detail that they must be aligned on a 4 byte boundary.

The Exec unit contains the utility functions `PasToC` and `CToPas` which convert between the two string formats by copying the data somewhere. For example,

```
PasToC('topaz.font', fontname);
```

The types `CString` and `pCString` are also defined in Exec.

Some commonly used functions such as `OpenLibrary`, `OpenDevice`, `OpenResource`, `OpenFont`, `Text` and `TextLength` automatically convert from a Pascal to a C string to allow you to say things like

```
OpenLibrary('icon.library', 33);
```

You should be aware that when using system structures, the responsibility for adding appropriate string conversions is upon the Pascal programmer.

Additional functions

Many useful operations are defined in the C headers as 'macros' which are expanded to code and inserted in your program each time they are used. All of these routines are provided as callable functions and procedures within the Pascal units.

Some additional functions can be found in the C linkable Amiga.lib library. This includes code for message port creation and initialising new list headers. A special unit called Amiga.unit (documented later in this section) provides these operations for Pascal programs.

Amiga.lib also contains absolute variables giving the address of certain hardware registers. These can be found for Pascal in the Hardware unit.

Name conflicts

In some cases, the conventional names used for record fields or functions conflict with Pascal reserved words or system functions. Such clashes were resolved by adding an underscore to the end of the offending name in the Amiga units. In all cases, it is the Amiga name which has changed and not the System unit or Pascal one. Such changes are listed alongside the unit documentation.

The most important example of this is the AmigaDOS unit which contains many similar functions. For example, `Read` refers to the System unit `Read` procedure whereas `Read_` is part of the AmigaDOS unit.

Another case is where an Amiga operating system unit and a Turbo Pascal unit use the same name. Since it is not advisable to mix use of these two unit types, these names have not been changed and you must manually prefix the unit name before the identifier, e.g.

```
Graph.ViewPort
```

The Intuition unit contains several cases where there are a function and constant of the same name. A trailing underscore can be used to specify the *constant*.

Libraries

Most of the Amiga system software resides in shared libraries. A library is simple a collection of functions which share a similar purpose. The corresponding HighSpeed Pascal unit gives you access to these functions and contains the relevant type definitions necessary for their use.

However, before you may use any library functions, it is necessary to 'open' the library. When your Pascal program starts it already has access to two libraries: DOS and Exec. The latter of these may then be used to open further libraries. Here is an example program which opens the Intuition library:

```
program LibraryExample;
uses Exec, Intuition;
begin
     IntuitionBase :=
pIntuitionBase(OpenLibrary('intuition.library', 33));
     if IntuitionBase = NIL then begin
          WriteLn('Unable to open Intuition library');
          exit
     end;

     {...main body of program, e.g...}
     DisplayBeep(NIL);

     CloseLibrary(pLibrary(IntuitionBase))
end.
```

Some things to note are the library name which is in lower case and the version number (33) which corresponds to release 1.2 of the system software, the earliest version still in use.

The `OpenLibrary` function of Exec actually returns the type pLibrary, i.e. a pointer to a library base. This data structure contains information concerning the library name and revision. The library functions are actually found in a 'jump table' which extends backwards in memory from the library base. It is to access this jump table that the library must be opened and assigned to the correct base name before calling its functions. Note also that `IntuitionBase` is not defined in your own program, it is actually contained within the Intuition unit.

Many libraries use this generic base structure although others, such as Intuition, have an extended library base structure which we must type explicitly. An extended library base is used by the library itself to store data. In many cases this data is private and should not be accessed by programs unless specifically documented, i.e. it is subject to change and may not work as you expect it to.

Another thing to note about the example program is the error checking and cleanup. On the Amiga you must always check that you obtain a resource which you tried to open. Even an `OpenLibrary` on a ROM-resident library may fail in certain low memory situations. If `OpenLibrary` returns `NIL` then you must close any other resources already opened and exit gracefully, preferably giving the user a meaningful error message. You must remember to `CloseLibrary` each library successfully opened before termination.

The following documentation gives for each library the correct library base name, corresponding Commodore C header files and a description of the library's function. A general overview of the library is then given. For a full description, please refer to the *Amiga ROM Kernel Manuals: Libraries.*

AmigaDOS

Basename:	`DOSBase` (opened automatically)
Headers:	libraries/dos.h, dosextens.h, dos_lib.h
Name Changes:	`Close_()`, `Delay_()`, `Exit_()`, `Input_()`, `Output_()`, `Read_()`, `Rename_()`, `Seek_()`, `Write_()`

Not to be confused with the Turbo Pascal compatible DOS unit, AmigaDOS is the interface to the lower level DOS library. The DOS library contains functions to examine and manipulate files as well as some utility routines and new process handling.

Files

Some of the key routines in the DOS library are `Open`, `Input` and `Output` which return file handles for subsequent `Read`, `Write` and possible `Seek` calls. You must `Close` any file handles which belong to you before your program terminates. If a DOS call fails, you may obtain an AmigaDOS error code from the `IoErr` function.

Opening a DOS device such as PRT: or SER: is allowed and a specification such as CON:0/0/640/200/Title can be used to open a new console window. `Input` and `Output` return the default file handles provided by DOS for your program. These often refer to the Shell's console window but may be re-directed the user. You should be aware that a program started from Workbench has no default input or output handles.

File `Locks` may be obtained from the `Lock` or `DupLock` functions and may be used to uniquely identify a volume, directory or file. `CurrentDir` accepts a lock referring to a new directory and returns the previous current directory. `Examine` and `ExamineNext` provide information on locked files. Remember to `UnLock` file locks once you have finished with them.

BCPL

AmigaDOS uses some conventions of the BCPL language in which it was originally written. Firstly, BCPL pointers are 'typeless' and are shifted right by 2 bits compared to the Pascal or C equivalents. This means that you must first find the pointer address with the `BADDR` function and then manually convert it to the appropriate type. For example,

```
cli := pCommandLineInterface(BADDR(process^.pr_CLI));
```

When passing memory to a DOS function, such as a `FileInfoBlock` to `Examine`, it must lie on a 4 byte boundary. If a BPTR is required within a structure you must also use `MKBADDR`. This alignment is guaranteed by both the HighSpeed Pascal and Exec library memory allocation schemes. BCPL strings are often used and, although similar to Pascal strings with a single length byte followed by the data, must also be aligned in this way.

For those interested in the structure of AmigaDOS, many of the DOS library functions simply form an interface to the underlying file systems. Calling the `Read` function for example will fill in a standard message packet with the arguments you supply, send the message to the appropriate filing system and wait for a reply.

DiskFont

Basename: DiskFontBase

Headers: libraries/diskfont.h

The DiskFont library is a disk-loaded library responsible for loading new fonts into the system. Using the OpenDiskFont function rather than the Graphics library's OpenFont function will automatically load a font from the current FONTS: directory if necessary.

Note that any number of programs may be using the same font in memory and even once each of these has called CloseFont, the data may still remain in memory. However, all unused fonts, libraries and devices will be flushed from memory once the system requires the memory for something else.

Exec

Basename: SysBase (opened automatically)

Headers: exec/alerts.h, devices.h, errors.h, exec.h, execbase.h, execname.h, exec_lib.h, interrupts.h, io.h, libraries.h, lists.h, memory.h, nodes.h, ports.h, resident.h, semaphores.h, tasks.h, types.h

Name Changes: TRUE_, FALSE_, tMemList, Insert_(), FreeMem_()

The Exec library (also known as the system executive) effectively coordinates the entire system. It handles task switching, library and device management, memory allocation, message passing, inter-task communication and interrupts as well as providing many low-level utility routines for manipulating and sorting linked lists, queues and stacks.

Lists

In order to keep track of the entire system, Exec makes extensive use of doubly linked lists via the `List` and `Node` records. The `List` header is slightly unusual in that it effectively contains a dummy head and tail `Node` which do not form part of the list. It also contains some information concerning the nature of the list. A list structure may be initialised with the supplied `NewList` function.

The `Node` record contains pointers to the previous and next nodes along with optional node type, priority and name. This record is often the first member of a larger structure. `MinList` and `MinNodes` provide the functionality of `Lists` and `Nodes` without any type, priority or name.

You may add and remove list nodes using `Insert`, `AddHead`, `AddTail`, `Remove`, `RemHead` and `RemTail`. `Enqueue` inserts a node according to its priority. `FindName` searches a list for the named node. In order to scan a list yourself, you must remember to ignore the head and tail nodes. Valid nodes always have successor and predecessors, never `NIL` pointers.

Memory

Memory may be allocated via `AllocMem` and `FreeMem_`. Provision is made for allocating Chip memory (accessible by the Amiga custom chips).

Special care should be taken when using the `MemList` record type. Unlike the C equivalent, it does not include an embedded `MemEntry` array. Instead you must define your own record type consisting of a `MemList` followed by the required number of MemEntries.

Messages

In order to use message passing you must first have a Port. This may be obtained from the system, such as an Intuition window's `UserPort` or from the `CreatePort` function of Amiga unit. `Wait`, `WaitPort`, `GetMsg`, `PutMsg` and `ReplyMsg` can then be used communicate via this port. The device I/O mechanism, Intuition's `IDCMP` messages and AmigaDOS's packet sending are all built on top of these basic functions.

To communicate with a device it is necessary to allocate and initialise an IORequest record using Amiga unit's `CreateExtIO`. `OpenDevice` will then fill in the device information and you must set up the `io_Command` and possibly `io_Data` and `io_Length` before calling `DoIO` (synchronous) or `SendIO` (asynchronous). `BeginIO`, `WaitIO`, `CheckIO` and `AbortIO` are also available for asynchronous device I/O. `CloseDevice` and `DeleteIOReq` or `DeleteExtIOReq` must be used before program termination.

Expansion

Basename:	`ExpansionBase`
Headers:	libraries/configregs.h, configvars.h, expansion.h, expansionbase.h

The Expansion library is the software that interfaces to the Auto-Config mechanism of the Amiga which allocates address space for controller and memory cards. It is only of use when writing new device drivers.

Graphics

Basename:	`GfxBase`
Headers:	graphics/clip.h, collide.h, copper.h, display.h, gels.h, gfx.h, gfxbase.h, gfxmacros.h, graphint.h, rastport.h, regions.h, sprite.h, text.h, view.h
Name Changes:	`tColorMap.Type_`, `Move()`, `Text_()`, `CInit_()`, `CWait_()` (`CINIT` and `CWAIT` refer to the more common macro equivalents)

There are basically three areas covered by the Graphics library: display control, drawing and animation. It is used by Intuition for all rendering and is closely linked with the Layers library.

Display

The graphics display primitives make use of `ViewPorts` which describe a horizontal strip of the display. `ViewPorts` are used by Intuition to display draggable screens with different modes and colour palettes. A number of `ViewPorts` may be chained together in one `View` which describes the complete display.

Display data is described in a `BitMap` and consists of one or more 'bitplanes' which represent a rectangular array of pixels. In order to represent different pen colours these bitplanes are effectively stacked with the corresponding pixel bits making up a pen number. The actual colour in which each pen is displayed is determined by the palette held in a `ColorMap` record.

`InitBitMap`, `AllocRaster`, `GetColorMap`, `InitVPort` and `InitView` initialise the structures and `MakeVPort`, `MrgCop` and `LoadView` generate the 'copper list' which coordinates the display hardware to produce the correct image. These resources must later be deallocated via `FreeRaster`, `FreeColorMap`, `FreeVPortCopLists` and `FreeCprList`.

Drawing

Drawing takes place via a `RastPort`. The `RastPort` contains information concerning the current co-ordinate positions, pen colours, drawing mode, text font, line drawing pattern and graphics layer (used for clipping). Some commonly used drawing procedures are `Move`, `Draw`, `DrawCircle`, `DrawEllipse`, `Flood` and `Text`. `SetAPen` and `SetBPen` are used to select foreground and background pen colours whilst `SetDrMd` sets up a drawing mode.

Many graphics operations work on a whole block of the display area. Examples of this are scrolling, drawing rectangles, moving images and clearing the display. The Amiga 'blitter' (block image transfer) chip may be utilised via the Graphics library; the calls `ScrollRaster`, `RectFill`, `SetRast` and `ClipBlit` provide this. More control over the blit operation is provided by the wonderfully named `BltBitMap`, `BltPattern`, `BltTemplate`, `BltBitMapRastPort` and `BltMaskBitMapRastPort`!

All of the graphics drawing routines may be used on the RastPort of an Intuition window. The system will handle all clipping so that only the visible parts of your window are drawn into (although you must be careful not to draw all over the window borders).

Animation

The animation routines provide facilities for controlling 'sprites' (images which move around the display). Simple sprites correspond to hardware sprites such as the one used for the mouse pointer. These are used as the basis of VSprites (virtual sprites) which make effective use of the available hardware sprites. These images are never actually drawn in the RastPort, instead they are superimposed over it by the graphics hardware.

Another form of movable image is the bob (blitter object). These are drawn by software and consequently are slower but give more flexibility. Workbench attaches bobs to the mouse pointer in order to provide draggable icons.

Animated images are controlled via an AnimComp (animation component) which are linked together to provide an AnimOb (animation object). Any of these structures may form a GEL (graphics element) which may be moved around the display at different velocities with acceleration control.

Icon

Basename: IconBase

Headers: workbench/icon.h

The Icon library provides application access to the .info files used by the Workbench. The GetDiskObject function loads the icon for a specified file and returns a pointer to a DiskObject record defined in the Workbench unit. In general, programs should only add to or examine an icons tooltypes and not adjust the imagery or default tool. The FindToolType and MatchToolValue functions may be used for this purpose.

To create or save an icon, PutDiskObject must be called. The icon may be one previously obtained from GetDiskObject or simply hard-coded into the program. To dispose of a DiskObject structure, call FreeDiskObject although care must be taken to restore any tooltypes which have been changed.

This library is disk loaded.

IFFParse

Basename: IFFParseBase

Headers: libraries/iffparse.h

IFFParse is a new library added with Release 2 of the operating system but also available under 1.3. It handles the reading and writing of IFF (interchange file format) files. IFFParse merely handles the job of 'understanding' the file format, interpreting or generating the file contents is completely up to you. The library has support for DOS files, the Amiga clipboard, user defined stream types and IFF property chunks.

The general method of using IFFParse is to first call AllocIFF to obtain the IFFHandle used to refer to the file. You can then use the AmigaDOS Open or IFFParse OpenClipboard function to initialise an iff_Stream and call InitIFFasDOS or InitIFFasClip. OpenIFF will then begin reading or writing.

By default, the ParseIFF function will simply read the entire file and tell you that it's done it. In order to do something useful you must first call StopChunk to inform the parser of which chunks you are interested in. PropChunk may be used to record property chunks which can be accessed via FindProp.

Writing IFF is performed through PushChunk and PopChunk. To begin writing a chunk, specify the name, ID and size (or IFFSIZE_UNKNOWN). WriteChunkBytes or WriteChunkRecords will actually output the data and PopChunk will complete the chunk, adding the correct size and pad byte if necessary.

For a full description of the use of IFFParse, refer to the *Amiga ROM Kernel Manuals: Libraries*. C language examples and the IFF specification may also be found in the *Devices* manual.

Intuition

Basename:	`IntuitionBase`
Headers:	intuition/intuition.h, intuitionbase.h, preferences.h, screens.h
Name Changes:	`CLOSE_, CLOSEWINDOW_, LONGINT_, REPORTMOUSE_, SHOWTITLE_, tStringInfo.LongInt_, tNewWindow.Type_, tNewScreen.Type_, Intuition_()`

Intuition is responsible for the windowing interface of the Amiga. It handles screens, windows, menus, gadgets and requesters and provides a high level mechanism for programs to receive mouse, keyboard and other system events. One thing which Intuition does *not* do is give scrolling text windows. For this you must use the console device.

Windows

Probably the most important object handled by Intuition is the window. A window can be opened by first creating and initialising a `NewWindow` record and passing it to the `OpenWindow` function. A pointer to a `Window` record will be returned and the `NewWindow` may then be freed or re-used. The `NewWindow` record contains information such as the window size, default title, system gadgets (close gadget, depth gadgets, sizing gadget etc.) and refresh type (simple, smart or superbitmap).

The `RPort` field of the `Window` points to a Graphics library `RastPort` which may be used for drawing into the window. The `Window` structure should generally be treated as read-only with adjustments to the window being carried out via the appropriate Intuition library functions. A program is responsible for closing each of its windows with `CloseWindow`.

Screens

All windows exist on Intuition screens. By default, the Workbench screen will be used but programs may open as many screens as they wish in a similar manner to opening windows. A `NewScreen` record must be supplied to `OpenScreen` which then returns a pointer to a `Screen`. This pointer can then be used for subsequent new windows and later closed with `CloseScreen`. The `Screen` structure contains `BitMap`, `ViewPort` and `ColorMap` records as used by `Graphics`.

Menus

A menu bar may be attached to a window once it has been opened via the `SetMenuStrip`. The `Menu` and `MenuItem` records are used to describe the menu bar, items and sub-items. You must call `ClearMenuStrip` before closing a window or attaching a new menu strip.

Gadgets

Intuition gadgets are the buttons, scroll bars and text input boxes used by the system. These are created by setting up a linked list of `Gadget` records each containing the position, size and type of a single gadget. The `BoolInfo`, `StringInfo` and `PropInfo` records contain further details for particular gadget types. The actual appearance of a gadget is determined by `Image`, `Border` and `IntuiText` records which are once again attached to the `Gadget` structure. These graphical objects are also used for rendering menus and Intuition provides support routines for using them from your own code.

Once initialised, gadgets must be added to a windows gadget list and refreshed before they appear on the display. Further manipulation is possible with `AddGadget`, `RemGadget`, `RefreshGList`, `GadgetOn` and `GadgetOff`. In order to determine the current state of a gadget you may examine certain fields in the relevant data structure.

IDCMP

In order to keep track of what the user is doing, your program must monitor each window's `IDCMP` (Intuition direct communication message port) for incoming `IntuiMessages`. When opening a window or calling `ModifyIDCMP` you may specify which events will be received for a given window. Keypresses, menu selections, gadget operations, mouse clicks or movements, disk insertions, resizes and close events are all reported in this way. Exec library's `Wait`, `GetMsg` and `ReplyMsg` functions are the usual method of handling these messages.

Your program may also be notified that its window requires 'refreshing', i.e. something has obscured part of its display which now needs redrawing. The system (through the Layers library) does provide a facility for automatic refreshing although it does involve some overhead. To do your own refreshing, simply call `BeginRefresh`, draw the entire window contents and call `EndRefresh`. This will optimise drawing so that only the damaged areas are drawn into. Intuition takes care of drawing the window gadgets itself.

It is only possible to give a brief overview of Intuition here but it is the section of the Amiga operating system which the applications programmer will spend much of his or her time working with. Indeed, large books have been written on the subject. Take time to familiarise yourself with its operation and try to stick to the standard user interface where possible.

Layers

Basename: LayersBase

Headers: graphics/layers.h

Co-operates with the Graphics library in order to provide overlapping 'layers' with automatic clipping and backup and restore of hidden areas. To quote the ROM Kernel Manual; "The Layers Library takes care of the mundane things, the low level, repetitive tasks that are needed to keep track of where to put which bits". When using Intuition, the operation of layers is mainly transparent to the programmer. Note that many layers functions actually form part of the Graphics library.

Essentially, a layer is a list of clipping rectangles associated with a `BitMap`. These `ClipRect` records form a graphics `Region`. Examples of `Regions` are the visible portion of a window, or the areas exposed by scrolling part of the display. Many utility routines for adding rectangles and regions together exist in the Graphics library.

Layers may be created, deleted, moved, resized, depth arranged and scrolled. It can be seen that these form the basis of Intuition's windowing system. Automatic backup of hidden areas for refreshing is available as is manual refreshing via `BeginUpdate` and `EndUpdate`. A layer may also use its own private `BitMap` which may be larger than the screen.

Translator

Basename: `TranslatorBase`

Headers: libraries/translator.h

Translator is a disk loaded library which essentially exists for a single function call; Translate. This converts a string of English language text into phonetics suitable for use by the Narrator device. This provides the text to speech conversion facilities of the Amiga.

Note that the AmigaDOS SPEAK: device handler provides a file-based interface to Translator and Narrator through the familiar `Open` and `Write` functions and thus you can use the Pascal `write` and `writeln` procedures to produce speech.

Devices

Amiga devices are used by the system for I/O. Each device must support a minimal set of commands including read, write, stop, start and flush. Whilst this is may be sufficient for a simple stream type device such as a serial driver, other devices require extensions to the basic model. Hence, many devices support 'non-standard' commands which are specific to that particular device.

Most devices exist as a separate task within the system, i.e. they run simultaneously with your program. Often, a device will support multiple 'units' such as a number of disk drives or audio channels. I/O is achieved by sending messages, I/O requests, to the device. Programs may either wait for their completion (synchronous I/O) or continue with other things while the request is being carried out (asynchronous I/O).

To use a device you must therefore have an understanding of the Exec library's message passing system. Messages require a 'reply port' which your program must provide although this may be shared between several devices or used for other purposes. I/O requests must also be correctly initialised which is handled by the Amiga unit function, `CreateExtIO`. Devices often use their own format of I/O request with special fields tagged onto the end.

Some devices, in addition to the message based commands described above, also provide directly callable functions, much like those given by libraries. In fact, the internal representation of a device is simply an extension of the Amiga library mechanism. Such devices require a base pointer to be initialised before their functions can be used.

One example of such a device is the Timer device:

```
program DeviceExample;
uses Timer, Exec, Amiga;
type ptimerequest = ^ttimerequest;
var req:   ptimerequest;
    port:  pMsgPort;
```

```
begin
    port := CreatePort(NIL, 0);
    if port = NIL then exit;

    req := ptimerequest(CreateExtIO(port,
sizeof(ttimerequest)));
    if req = NIL then begin
     DeletePort(port);
     exit
    end;

    if OpenDevice('timer.device', UNIT_VBLANK,
pIORequest(req), 0) <> 0 then begin
     DeleteExtIO(pIORequest(req));
     DeletePort(port);
     exit
    end;

    TimerBase := pLibrary(req^.tr_Node.io_Device);
    {...main body of program...}

    CloseDevice(pIORequest(req));
    DeleteExtIO(pIORequest(req));
    DeletePort(port);
end.
```

This illustrates the various setup required for device I/O. A message port is created. The I/O request block, in this case a `ttimerequest`, is allocated and initialised for use on that port. The correct device unit is opened using the I/O request block and finally, the device base pointer is copied into `TimerBase` in order to access the timer functions. With devices that do not have any basename, this last step is not necessary.

Note that, at each stage, the program checks for errors and exits gracefully if unsuccessful. It should be pointed out that the `OpenDevice` call, unlike `OpenLibrary`, returns zero for *success* or an appropriate error code. When using device functions it is often necessary to convert pointers into the appropriate generic type such as `pIORequest` before use.

In order to send an I/O request, the appropriate fields for the device must first be filled in. Often this is simply a matter of setting `io_Command` to the command number, `io_Data` to point to the data to be sent or receive buffer and `io_Length` to its size. The amount read or written will be returned in `io_Actual`. Other devices have different conventions. The following Exec functions are available for device I/O: `DoIO` (synchronous, waits for completion), `SendIO` (asynchronous), `CheckIO`, `WaitIO` and `AbortIO`.

After successfully opening a device, the `io_Device` and `io_Unit` fields may be copied into another suitable I/O request block in order to send multiple requests without having to reopen the device.

AmigaPrinter

Basename: none

Headers: devices/printer.h, prtbase.h, prtgfx.h

The AmigaPrinter unit allows direct access to the Printer device and should not be confused with the Turbo-Pascal compatible Printer unit. The Printer device is a disk loaded device which via different printer drivers provides translation of ANSI control sequences and full colour or monochrome screen dumps. Output is directed through SER: or PAR: according to current user preferences.

Note that for many text applications it is simpler to use the DOS printer device, PRT: through the `Open`, `Write` and `Close` functions. This method has the additional benefit of easy redirection to a file or other DOS device. In order to use either of these methods, the printer.device file must be in DEVS: with the printer driver selected in Preferences in DEVS:printers.

The Printer device uses 3 types of I/O requests; `IOStdReq` for `CMD_START` and `CMD_WRITE` etc., `IOPrtCmdReq` for sending printer commands directly (`PRD_PRTCOMMAND`) and `IODRPReq` to request a graphic dump of a `RastPort` (`PRD_DUMPRPORT`). These may be allocated with `CreateExtIO`.

The standard I/O commands all work as expected, translating ANSI command sequences where necessary. Note that if a printer driver does not support a particular command, it will simply be ignored. A common mistake is to send escape codes suitable for your printer rather than standard ANSI commands. See your system documentation for a complete list of the supported commands. The PRD_RAWWRITE I/O command sends unprocessed data to the printer.

One of the main reasons for using the Printer device directly is to utilise the graphic dump facilities. A sub-rectangle of a RastPort may be printed with the supplied ColourMap, ViewModes (including HAM, Extra Half Bright etc.), aspect ratio and page positioning. Note that the printer device is not limited by Amiga screen resolutions.

Audio

Basename: none

Headers: devices/audio.h

The Audio device handles sound output and arbitration for the system. It should be used even when writing to the hardware registers directly in order to avoid conflicts with other programs which may be running.

An IOAudio request block is used for audio device I/O. When OpenDevice is called, you may specify channels to be allocated via the ioa_Request.ioa_Length field in the same was as if you had send a ADCMD_ALLOCATE command. An ioa_Length of zero simply opens the device for later channel allocation. Many commands work on multiple audio channels determined by the io_Unit field.

The ADCMD_ALLOCATE command takes a sound precedence in io_Message.mn_Node.ln_Pri to determine whether it can override other sounds which are in progress. Sounds may be started with a CMD_WRITE after filling in the appropriate fields of the IOAudio request block.

The Amiga generates sounds from waveform or sample data which must be located in Chip memory. Even simple waveforms such as square, sine or sawtooth waves must be represented by an array of signed byte values.

Clipboard

Basename: none

Headers: devices/clipboard.h

The Amiga Clipboard provides a rendezvous for applications that wish to 'cut' and 'paste' data. It supports multiple clip units, IFF (interchange file format), disk spooling, random access and delayed 'post' write requests. It is highly recommended that all programs supporting block operations use the Clipboard.

Clipboard I/O uses an `IOClipReq` which is identical to a standard request with the addition of the `io_ClipID` field. Although the device supports 255 separate unit numbers (requested through `OpenDevice`), the `PRIMARY_CLIP` unit should be used unless requested through user preferences. Note that the clipboard.device file must be present in DEVS: for use.

The sequence of events for cutting a block to the clipboard is to `CreateExtIO` an `IOClipReq`, `OpenDevice` the appropriate unit, `CMD_WRITE` with `io_Offset` and `io_ClipID` set to zero. The `io_Data` and `io_Length` fields indicate the length of the data. Any number of `CMD_WRITEs` may be used, terminated by a `CMD_UPDATE` to complete the clip.

Pasting is achieved by sending `CMD_READS` (`io_Offset` must again be initialised to zero) until the returned `io_Actual` is zero, indicating end of clip.

In order to support the clipboard, a program must understand IFF. The most common IFF forms used are FTXT (formatted text) and ILBM (interleaved bitmap) for graphics. For a full description of the IFF standard, please refer to the *ROM Kernel Manuals: Devices, Third Edition.*

Console

Basename: `ConsoleDevice`

Headers: devices/console.h, conunit.h

The Console device provides ANSI terminal capabilities within an Intuition window. ANSI control sequences, line buffering and editing (as used by the CLI and Shell) are supported. The Console device may also be accessed in a file-based manner via CON: under AmigaDOS.

Opening a Console device involves creating a port, `IOStdReq` and Intuition window. An `OpenDevice` of unit 0 with `io_Data` and `io_Length` set up for the Window structure will fill in `io_Device` with the device base pointer and `io_Unit` as a pointer to a `ConUnit` record. Many useful values such as the cursor position and dimensions may be read from this data structure.

Characters are output to the window and read back using `CMD_WRITE` and `CMD_READ` in the usual way. Control sequences to request special event reporting or keyboard mapping information may also be sent in this way. Note that the console unit must be closed *before* the associated Intuition window.

In addition to I/O operations, the console device offers several library style functions such as `RawKeyConvert` to translate raw keycodes into text characters or control sequences. In order to use these, ConsoleBase must be initialised from a valid `io_Device`. Opening unit -1 of the device may be used to obtain this base pointer without opening a new console window.

Gameport

Basename: none

Headers: devices/gameport.h

Gameport is the device which handles mouse and digital joystick input. There is no system support for proportional joysticks. This information is in turn passed on to the Input device to form part of the event stream.

Two units are available, corresponding to the Amiga's two joystick ports. Unit 0 is normally used for the mouse leaving unit 1 available for other purposes. You may configure the Gameport device to respond to specific events from a particular type of controller. Each action is returned in the form of an `InputEvent`.

Input

Basename: none

Headers: devices/input.h, inputevent.h

The purpose of the Amiga Input device is to collect and organise the stream of user input events from the keyboard, mouse, disk drives etc. into what is sometimes referred to as the 'food chain'. Intuition and the Console device use this to react to user actions and programs may add to or adjust input events as required.

The most common reason for directly opening the Input device is to add 'hot keys' (keypresses which activate a program or operation regardless of which application is running) to the system. This is done by inserting a handler into the input chain. For other applications, it is usually preferable to use Intuition's `IDCMP` scheme or access the Console, GamePort or Timer devices.

Keyboard

Basename: none

Headers: devices/keyboard.h

The Keyboard device gives system access to the Amiga keyboard and is usually only accessed by the Input device. Commands to read the keyboard matrix and to add reset handlers for emergency processing upon user reset are also supported.

Warning: reading key events directly from the Keyboard device will prevent them from reaching the Input device and system event stream. Use Intuition or the Console or Input device instead.

Narrator

Basename: none

Headers: devices/narrator.h

The Amiga Narrator is the device which produces recognisable human-like speech from phonetic text. When used in conjunction with the Translator library, English language text to speech translation can be achieved. The Narrator can optionally give information on suitable mouth shapes for synchronised animation.

Narrator uses two forms of I/O request block; `narrator_rb` and `mouth_rb`. The first of these is filled in by the OpenDevice call and is used for `CMD_WRITE` commands. It contains details of the voice speed, pitch, sex, volume etc. The `io_Data` string must be terminated by the sequence `Q#U` and a zero byte and its length should be placed in `io_Length`.

The phonetic alphabet used is ARPABET, a subset of the International Phonetic Alphabet represented in ASCII. Each sound consists of one or two upper case letters which roughly correspond to the actual sound. Numbered stress values and punctuation are also recognised.

HighSpeed Pascal is supplied with a demonstration program which illustrates the usage of the Amiga's speech capabilities.

Parallel

Basename: none

Headers: devices/parallel.h

The Parallel device is used for bi-directional I/O via the Centronics compatible parallel port. It is used by the Amiga Printer device which should be employed for all printer output.

The `IOExtPar` request is used and is initialised upon calling `OpenDevice`. Standard commands can be used to read and write data. The device can be programmed to accept a number of 'end of transfer' characters if required.

SCSI

Basename: none

Headers: devices/scsidisk.h, hardblocks.h

SCSI (small computer system interface) hardware is used on the Amiga 3000 computer and A2091/A590 hard drives to transfer data under the control of the SCSI device. This device can cope with standard I/O commands as well as some Trackdisk commands and `HD_SCSICMD` to issue a SCSI-direct command to any SCSI unit on the bus.

The `RigidDiskBlock` structure used to provide automatic mounting of hard drive partitions also forms part of the HighSpeed Pascal SCSI unit. Note that different hard drives use other driver software and mount information. All file level access must be done through AmigaDOS and not through the device interface.

Serial

Basename: none

Headers: devices/serial.h

The Serial device controls the Amiga's built in RS-232C compatible serial port. It supports a variety of transfer protocols and baud rates. Note that output intended for a printer should instead be sent to the Amiga Printer device.

An `IOExtSer` request block contains the parameters required for serial transfer. This is achieved by use of standard device commands with some extra commands providing control over various parameters.

Some applications requiring high speed serial transfer, notably MIDI programs, may 'miss' data due to system software overhead. If it is necessary to write to the hardware directly, this should only be done after successfully allocating the appropriate bits through the Misc resource.

Timer

Basename: TimerBase

Headers: devices/timer.h

Provides general purpose timing through an Exec style device. Two timer resolutions are supported, vertical blank (through UNIT_VBLANK) for long duration requests and microhertz (UNIT_MICROHZ) for short burst timing.

The Timer device can also be used to read or set the system clock and for library routines involving time calculations based upon I/O requests.

The device is intended for interval timing and does not support stopwatch or alarm clock style functionality For general purpose application 'prods' for updating scroll bars or list displays, Intuition windows can provide 'intuitick' events at a rate of around 10 per second.

The Timer device uses a timerequest record for I/O which is simply an IORequest with fields for seconds and microseconds added to it. When sending timerequests to the timer device via SendIO, you must be careful not to re-use the request block until the device has replied to it. Any number of requests may be sent and they will be returned in the appropriate order. Using DoIO will simply delay your program for a period of time in a similar way to the AmigaDOS and HighSpeed Pascal Delay procedures.

In order to use the library functions of the Timer device you must first open the device and copy the io_Device field of your timerequest into the TimerBase variable. The timer arithmetic functions allow addition, subtraction and comparison of two timerequests.

One bug which exists in the Timer device under Kickstart 1.3 and before is that asking for a time delay of 0 or 1 microseconds can crash the system.

Trackdisk

Basename: none

Headers: devices/trackdisk.h

Trackdisk is the device used by the Amiga filing system to read and write floppy disks. It is intended for advanced users only. Some standard commands are supported and in addition, many device specific commands to control the motor, seek to a specific track, format the disk, check disk and drive status and read/write raw MFM data are available.

Resources

In general, resources handle arbitration of shared hardware. Since the Amiga multitasks, different programs or devices may require access to the hardware and to ensure they do not conflict, this is done through the appropriate resource.

Important note: resources are just one step above direct hardware manipulation. There is usually a higher level device or library interface that is easier to use, maintain and upgrade and which shares the hardware more efficiently between applications.

You must open a resource before calling its functions. The correct basenames are listed alongside the resource description. An example of this is:

```
program ResourceExample;
uses CIA, Exec;
begin
     CIABase := pLibrary(OpenResource('ciab.resource'));
     if CIABase = NIL then exit;
     {...main body of program...}
end.
```

You may notice that there is no corresponding `CloseResource` call. Unlike with libraries and devices, this is not necessary.

CIA

Basename: CIABase

Headers: resources/cia.h

This resource grants access to the interrupts and timer bits of the 8520 CIA (complex interface adaptor) chips. Some applications such as MIDI and time coding programs require such extreme accuracy and low overhead timing. However, the vast majority of programs should use the Timer device.

There are only four interval timers available and some of these will already be in use by the system. The correct method of allocating a timer is to add a timer interrupt using `AddICRVector`. There are two CIA chips with resource names of `ciaa.resource` and `ciab.resource`. Normally, you should open `ciab.resource` and attempt to allocate timer B, or timer A if it was already in use.

Disk

Basename: DiskBase

Headers: resources/disk.h

The Amiga supports up to 4 floppy disk units which may be reserved through use of the Disk resource. Routines exist to permanently allocate or temporarily 'borrow' a disk unit although you should be aware that the Trackdisk device normally allocates all units for its own use.

FileSystem

Basename: none

Headers: libraries/filehandler.h, .i.,
devices/bootblock.h,
resources/filesysres.h

As has been mentioned, the AmigaDOS filing system is built upon a number of 'file handlers' which communicate via a standard packet based interface. The two most common file handlers used by floppy and hard disks are the Fast File System (FFS) and Old File System (OFS).

The FileSystem unit contains a number of structures and definitions which are used by the system to determine and update the current configuration. A list of available file systems may be obtained by opening the FileSystem resource.

Misc

Basename: MiscBase

Headers: resources/misc.h

The Misc resource is used to arbitrate the use of several important registers which control the serial and parallel hardware. It should be used when a program wishes to drive such hardware directly rather than through the standard device interface.

The correct way to allocate these bits is to call MR_ALLOCMISCRESOURCE with the appropriate constant and an identification name. If the call returns NIL, you now have exclusive access until MR_FREEMISCRESOURCE is called.

Note that all versions of the Serial device up to and including 1.3 contain a bug which prevents them from correctly freeing the serial bits when closed but still resident in memory. The workaround for this is to Forbid, FindName the serial device on the ExecBase device list, RemDevice it and Permit.

Potgo

Basename: PotgoBase

Headers: resources/potgo.h

The Potgo resource allows control of the hardware POTGO register connected to some I/O pins on the joystick ports. The Gameport device and Intuition provide other methods of reading joysticks and mouse buttons. The functions AllocPotBits, FreePotBits and WritePotgo should be used to access these bits.

Other

Some of the Amiga HighSpeed Pascal units do not fall under the category of libraries, devices or resources. Probably the most important one of these is the Amiga unit which contains many utility routines necessary for device I/O. The others are collections of constant and type declarations used by the system.

In order to use these units you simply include them in the Uses clause at the start of your program. No other preparation is required.

Amiga

Headers: none

This contains utility functions normally found in Commodore's linkable Amiga.lib library. The functions CreatePort, DeletePort, CreateExtIO and DeleteExtIO are necessary for message passing and device I/O. NewList is the procedure used to initialise an Exec List record to represent an empty list.

CreatePort takes an optional name and port priority and returns type pMsgPort or NIL if it fails. If the name parameter is not NIL, the port will be added to the system's public port list which may be accessed by other programs. This is not necessary for most ports and should only be used for inter-program communication. Similarly, a program's own MsgPorts will normally have a priority of zero.

`DeletePort` must be used for a `MsgPort` created with `CreatePort`. It will automatically remove a named port from the public port list. You must take care that all messages are completed and have been replied to before deleting the port.

`CreateExtIO` takes a pMsgPort and an I/O request size, returning the generic type `pIORequest` or `NIL` if it failed. It is used to allocate any I/O request block and correctly initialise the standard information at the start of it for use with the given port.

`DeleteExtIO` is necessary to deallocate an I/O request block. It should only be called once the request has been completely finished with and replied to.

Hardware

Headers: hardware/adkbits.h, blit.h, cia.h, custom.h, dmabits.h, intbits.h

Name Changes: `tbltnode.function_`

The Hardware unit is a collection of definitions necessary when directly reading and writing the Amiga hardware registers. Whilst the operating system is running you should never do this unless you have gained exclusive access to the hardware through the correct system functions. Failure to do so can result in unexpected crashes, loss of data, hard disks etc. You have been warned!

Essentially, the `custom`, `ciaa` and `ciab` variables hold the absolute addresses of these hardware register blocks. These are defined as pointers to `tCustom` and `tCIA` record types which give the actual register names.

You should be aware that some hardware registers trigger actions upon read or write accesses. Great care must be taken with these registers to ensure the correct behaviour. Since behaviour may change between versions of the compiler it is safest to use the inline assembler for such purposes.

Keymap

Headers: devices/keymap.h

This unit defines the `Keymap` record type which contains all the necessary information to translate physical key presses into ASCII sequences. This information is used by the Console device and Intuition to translate raw key codes from the hardware via Input device into readable characters and cursor control sequences.

Note that the Console device supports commands to read and write the keymap associated with a given console window and the system default keymap.

Workbench

Headers: workbench/startup.h, workbench.h

The Workbench unit gives the definition of a DiskObject as used for Workbench icons and supported by the Icon library. For further details refer to the library description.

C for Pascal programmers

Users programming the Amiga in any computer language will discover that most of the documentation, books and example programs which exist are based around the C language. This is partly because much of the machine's operating system was originally written in C and partly because of the language's current popularity and the early availability of an Amiga C compiler.

We have gone to a great deal of effort to ensure that HighSpeed Pascal supports the Amiga in a comprehensive and natural manner and is in every respect as capable as C of producing superb application programs. However, whether you are using Pascal because it is the language which you are used to or simply because you find it more tasteful, you will find a basic knowledge of C most useful when trying to use the Amiga to its full.

In the following section we will attempt to furnish the reader with a basic working knowledge of C, enough to understand the system documentation and example programs. We would like to stress that it is by no means intended to *teach* C to Pascal programmers (why would we want to do such a thing?), merely to provide enough information to allow you to figure it out for yourself.

Data structures

Basic types

Like most languages, C provides a number of basic types which represent characters and numbers; these may be employed for variables, constants, structured types etc. You can use this section for converting C types to Pascal.

Here is a list of them and their HighSpeed Pascal equivalents:

C type	HighSpeed type	byte size
`char`	`ShortInt`	1
`int`	see below	2/4
`short`	`Integer`	2
`long`	`LongInt`	4
`unsigned char`	`Byte/Char`	1
`unsigned int`	see below	2/4
`unsigned short`	`Word`	2
`unsigned long`	no equivalent	4

The first thing to notice is that `int` and `unsigned int` can have two sizes. This is because the type `int` is used to specify the *natural* integer type. It may be either a long or a short integer. In practice, most Amiga programs are written for long integers only but you cannot always assume that this is the case.

Another point of confusion is that both signed and unsigned character types exist. Each of these can be used for storing numbers or ASCII characters. Although `char` is supposed to mean a signed quantity, C compilers often give the option of making it default to unsigned to avoid bizarre problems when converting to other types.

C lacks a boolean type, the type `int` is often used instead. To remove this and other ambiguities, Commodore defined a number of types which are used extensively in the system documentation. They are as follows.

Commodore type	HighSpeed type	byte size
`BYTE`	`ShortInt`	1
`UBYTE`	`Byte`	1
`WORD`	`Integer`	2
`UWORD`	`Word`	2
`LONG`	`LongInt`	4
`ULONG`	no equivalent	4
`BOOL`	`Boolean/Integer`	2

As you can see, these will probably confuse you even more because `WORD` is actually an Integer and not a Pascal Word at all! However, it is not all bad because the Amiga Pascal units don't use these types. Instead they have been translated into the correct Pascal equivalents although you will need to understand them when looking at C code.

HighSpeed Pascal does not provide an unsigned long type. Instead, you will have to make do with `LongInt`. In practice, this does not present too much of a problem as you may specify hex numbers to allow use of the full 32 bits. Unsigned long is often used for bit flags.

Constants

In general, constant values look much the same in both languages. One important distinction is between character and string constants which use different delimiters. These and other constants are illustrated below.

type of constant	example
decimal	123
signed number	-287
decimal long	96L
hex	0x1800
octal	0375
character	'a'
string	"hello"

In addition to this, there are some special rules for character constants and strings. The backslash character (\) is used to denote certain special control sequences.

You should also be aware that strings may be split over several lines without using any special concatenation character.

character sequence	represents
\n	end of line
\r	carriage return
\l	line feed
\t	tab
\f	form feed
\0	end of string
\D	decimal character code
\xDD	hex character code

Although C has a `const` keyword to define constants, most programs make use of the `#define` directive of the *preprocessor*. The preprocessor acts like an extra pass of the compiler which goes around substituting defined values or names and including other files. Although `#define` is capable of great sophistication, you will often see it used to specify a numeric or string constant as in

```
#define VANILLAKEY 0x00200000
```

Commodore often use the convention of defining a flag bit number and value with `B` and `F`, e.g. `MEMB_CLEAR` refers to the bit number whilst `MEMF_CLEAR` is the actual value you would use in expressions.

Another method of defining a list of related constants in C is an enumeration. For example

```
enum Colours { Red, Green, Blue };
```

is equivalent to the Pascal

```
Type Colours = (Red, Green, Blue);
```

This type may be referred to as `enum Colours` and is equivalent to `int`.

Pointers

Now we are getting to more interesting things. C supports typed and untyped pointers. A pointer of unknown type is known as a 'void' pointer. This is similar to Pascal's generic `Pointer` type and relaxes type checking somewhat. Pointers are denoted by the * character. For example,

```
void *something;
char *name;
```

In the second example, the star means that it is a character *pointer* in a similar way to the Pascal ^. A pointer to a pointer is declared as

```
int **table;
```

The star character is also used as a 'dereferencing' operator, i.e. it gives you what the pointer is actually pointing at. In this capacity it works like a ^ after the name in Pascal. For example,

```
initial = *name;
```

would put the character pointer to by name into the variable initial. Note that the equals sign in C represents an *assignment* rather than equivalence as it does in Pascal.

The opposite of the * operator is & or 'address of'. It gives you a pointer to the object rather than its value and is used in much the same way as Pascal's @ operator. The symbol `NULL` is often used to specify a pointer that is not pointing anywhere. Its actual value, like `NIL` in Pascal, is zero.

The Amiga header files from Commodore define several pointer types used by the system.

pointer type	purpose
APTR	a void pointer
BPTR	an AmigaDOS BCPL pointer, shifted right by 2 bits
CPTR	general purpose pointer
STRPTR	pointer to a C null-terminated string

An unusual feature of C pointers is that adding and subtracting numbers from them gives you a pointer to the next or previous object, i.e. adding one to a long pointer makes it point to the next *longword* in much the same way as adding one to a character pointer causes it to point to the next character. This is how arrays are implemented.

Arrays

Pointers and arrays in C are very closely linked. Defining an array of 10 integers is a matter of saying

```
int array[10];
```

There is no lower bound specified as with Pascal because C always starts array numbering at zero. The expression `array[9]` would thus give the last element in the array. Because of the way arrays work, just saying `array` actually yields a pointer to the start of the array and `array+3` is a pointer to the fourth (numbering starts from zero, remember) element of it!

It is possible to miss out the number of elements in an array declaration. This can be very dangerous as the compiler can no longer warn you if you are writing to array elements that don't actually exist but it is a powerful technique. It allows you to have arrays of no fixed size, dynamic arrays, or to determine the array dimensions at run time.

It is this laxness of type checking that permits C programmers to miss out the [] characters on many occasions, use a simple character pointer to point to a string and to say 'pointer' when they really mean 'array'. Beware.

Multi-dimensional arrays are used in the following way.

```
short minutes[24][60];
minutes[12][0] = minutes[23][59];
```

In other words, a 24-deep array where each element is itself an array of 60 short integers.

Pointers and arrays are one of C's most powerful features. Although confusing at first, they are very useful. The following comparison table may help to clear up a few points.

C expression	Pascal equivalent	comment
`*name`	`name^`	value which name is pointing to
`short *numbers;`	`var numbers: ^integer;`	pointer declaration
`unsigned char id[4];`	`var id: array [0..3] of byte;`	array declaration
`grid[8][12]`	`grid[8,12]`	using a multi-dimensional array
`*messages[2]`	`messages[2]^`	extracting value via a table of pointers

Structures and Unions

These are simply the equivalents of fixed and variant records. A structure may contain simple types, pointers, arrays and have other structures embedded within it. To demonstrate this, here are C and Pascal declarations of a similar structure.

```
struct student {
   struct Tutor *next;
   struct History info;
   char name [32];
   int age, stay;
};
```

```
Type student: Record
   next: ^Tutor;
   info: History;
   name: Array [0..31] of Char;
   age, stay: Integer;
End;
```

As you can see, C uses the reverse order for declaring member names, putting the type before the identifier. Note the star in the first element which denotes a pointer to a different structure whereas the next member actually includes an instance of a structure.

The structure name, or 'tag' as it is known in C, does not form a type definition. That is, it must always be preceded with the word `struct` to refer to that structure unless a `typedef` is used (see the next section). A tagless `struct` may be defined (inside another `struct` for example) although it can never again be referenced.

Unions follow a very similar syntax although instead of each member being placed sequentially in memory they are overlaid on top of one another.

Like variant records this is used to represent data which may be accessed in a number of ways or a structure which may contain different sets of values depending upon the circumstances. Note that there is no C equivalent for the Pascal discriminator or variant selector, programmers must implement this manually.

Type checking

Pascal and C have fundamentally different ways of looking at types. Pascal is a 'strongly typed' language which can discriminate between two differently named integer types. C on the other hand is quite happy-go-lucky about the whole issue and allows programmers to move just about anything to anywhere.

As has already been mentioned, a `typedef` may be used for structures to save having to add the `struct` keyword. This construct may also be used on simple types and arrays although its use is mainly cosmetic as all `typedef`s are simply converted back. For example,

```
typedef struct Student student_type;
typedef struct Student *student_ptr;
typedef int year_of_birth;
typedef short[3][3] matrix;
typedef struct {int x,y} point;
```

The last one of these defines a tagless structure with the type name point. You can define a structure and type of the same name.

On occasions where you need to convert from one type to another (often caused by functions which use generic pointers and types - allocating memory for example) C allows you to specify a 'coercion'. Simply placing the type name in brackets before the value performs this.

```
button = (struct Gadget *)
     AllocMem(sizeof(struct Gadget),
     MEMF_CLEAR | MEMF_PUBLIC);
```

Here, the void pointer returned by `AllocMem` is being explicitly typed to a gadget pointer before assignment to button. By contrast, the Pascal way of doing things is to use the type name as if it were a function call.

There are many cases where C programs get away with this because the compiler automatically converts between integer types.

Programs

Functions

The declaration of a function in C looks very much like the Pascal version. Function prototypes are often used which look like a function definition without any code following it, just like the declarations in the interface section of a unit. Here is an example which demonstrates the layout of a C function:

```
int main(int argc, char *argv[])
{
     int counter;
     char digits[] = {1, 2, 3};
     return 5;
}
```

The function return type is given first followed by the name and bracketed parameter list. The braces or 'curly brackets' are the C equivalent of the `Begin` and `End` keywords. You will see a lot of these. It's often easier just to ignore them all when trying to understand a C program!

Any local variables associated with a function come immediately after the opening bracket and may be initialised after an equals sign. The previous example shows initialisation of an array using more curly brackets, structures can be initialised in the same way.

Unlike Pascal, C makes little distinction between a procedure and a function. In C, a procedure is known as a `void` function, i.e. a function with no return value. Similarly, a function which takes no parameters is said to have a void parameter list. A simple example of this is:

```
void simple_function(void);
```

For functions which do return a value, the `return` statement may be used to exit the function with the given number. Any number of returns may appear in a single function and no return value is necessary for void functions. The `exit()` function will leave the entire program.

`Main` is a special function in C. It is the function run when the program starts up. By convention, it takes two parameters: `argc`, the argument count and `argv`, an array of string pointers for each command line argument passed to the program ending with a null pointer. These can be obtained in HighSpeed Pascal via the System unit's `ParamCount` and `ParamStr` functions. `Main` returns an `int` which is an error code or zero for success.

C also allows functions with a variable number of arguments. This is denoted in the command declaration by three dots. Suffice it to say that subsequent parameters are accessed via pointers and that no type checking is carried out. The most used command of this ilk is `printf()` detailed towards the end of this section.

If a C programmer decides that hs or she is not interested in the return value of a function, they can call it just as if it was a void function or procedure call. This is often used by lazy people who don't check to see if an error has occurred.

Macros

As has been mentioned, C uses a preprocessor which can be used for simple text substitution or defining constants. However, it does have a facility for parameter substitution. This is usually used for short sequences which are tedious to enter time and time again but which do not warrant a function definition. An example of this might be

```
#define RECSIZE(n) (n * sizeof(Record) + sizeof(Header))
```

When referred to via `RECSIZE(10)`, the preprocessor will substitute the calculation with `n` equal to the value 10. In effect, such macros are mini functions and you may treat them as such.

Advanced macro programming is something of a black art with additional operators and rules applying. However, in general if you implement a macro as a Pascal function or procedure (as is done in the HighSpeed Pascal Amiga operating system units) you will get along fine.

Expressions

The diversity of operators in C and their precedence is a source of great confusion, even (or especially, depending upon your point of view) for C programmers. Let's start off with a list of all the operators and their Pascal equivalents. The following table is in the C order of precedence, i.e. the ones at the top are done first. Don't worry about the gaps on the Pascal side; we'll come to that later.

C operators	Pascal operators
`() [] -> .`	`() [] ^. .`
`! ~ ++ -- + - * & (type) sizeof`	`not + - ^ @ type() sizeof`
`* / %`	`* mul / div mod`
`+ -`	`+ -`
`<< >>`	`shl shr`
`< <= > >=`	`< <= > >=`
`== !=`	`= <>`
`&`	`and`
`^`	`xor`
`|`	`or`
`&&`	
`||`	
`?:`	
`= += -= *= /= %= &= ^= |= <<= >>=`	`:=` (not an operator)
`,`	

As you can see, many of these are simply a matter of using different symbols. However, there are some major differences. Taking it from the top, the `++` and `--` operators are a way of changing a value by one. They are both an operator and an assignment and have the unusual property of returning a different value depending upon the order. For example, `a++` gives the value of `a` before incrementing it whereas `++a` increments `a` and *then* gives its value.

When combined with pointers, ++ and -- can be used as follows:

```
nextchar = *name_ptr++;
sameagain = *--name_ptr;
```

These statements return a character via the pointer, updating it to point to the next or previous characters. This implements post-increment and pre-decrement stacks.

Note that C has a single multiply and divide operator rather than different ones for integers and floating point.

The bitwise operators (&, |, ^) are identical to `and`, `or`, `xor` in Pascal but C also gives *logical* operators. These guarantee a certain sequence of execution which must be simulated with `if` statements in Pascal. Logical 'and' (&&) says that the second part will only be executed if the first part is non-zero. Logical 'or' (||) doesn't bother with the right hand expression if it has already found that the first part is false.

The logical operators can be used to great effect (in traditionally cryptic style) if function calls form part of the expressions. In the following example, no data will be written if `ReadData` returned FALSE.

```
if (ReadData(source, buffer) != FALSE &&
     WriteData(dest, buffer) != FALSE)
          printf("OK\n");
```

The entry shown as ?: in the table is in fact two parts of a single construct sometimes known as the tertiary operator. In fact, it is a short circuit form of the 'if' statement which takes three parts; a condition, a true part and a false part. To clarify, the statement

```
value = (inputch < 10) ? inputch : 10;
```

is equivalent to the Pascal

```
if (inputch < 10) then
     value = inputch
else
     value = 10;
```

It may surprise you to see = (assignment) listed as an operator (remember, this is different from the Pascal = which tests for equality like == does in C). Equals is an operator because it returns a value, the value assigned. This means you can string them together as in `a = b = c = 99`. You can even start putting them inside expressions and conditional expressions, e.g.

```
if ((mem = malloc(100)) == NULL) panic();
```

which calls `malloc` with the number 100, assigns the return value to the variable mem and tests if equal to `NULL`. If used sensibly, this practice can condense several lines of code but taken to extremes it makes many complex expressions difficult to read at a glance.

There are a number of flavours of the assignment operator which add a number to a variable and suchlike. In C you may say `value += 10` rather than `value := value + 10` or `inc(value,10)`.

Finally, the comma operator is used to separate multiple expressions in much the same way as the semicolon separates statements. In effect, it returns the value of the rightmost expression.

Control structures

Let's start with the `if` statement in both languages.

```
if (a < b)
    do_something();
else
    do_something_else();
```

```
if a < b then
    do_something
else
    do_something_else;
```

Note the parentheses round the if expression,.the lack of `then` and the extra semicolon. Don't go typing this into your Pascal program because it will get confused!

A semicolon in Pascal is a separator whilst in C you just put one on the end of every line (well, almost).

A more complex `if` statement would use more than one statement in each case (shown by more curly brackets) and possible a number of ifs strung together, as in

```
if (opened)
{
     LoadPicture(file, buffer);
     Display(picture);
     count += 1;
}
else if (error != NOT_FOUND)
     ErrorRequester();
else
{
     printf("What picture?\n");
     return 20;
}
```

Very similar looping structures exist in both Pascal and C. While loops are almost identical and `do..while` loops can be translated into `Repeat..Untils` such as

```
do
{
    c = getch(stream);
    counter++;
} while (c == ' ');
```

```
Repeat
    c := getch(stream);
    counter := counter + 1
Until c <> ' ';
```

The statements `continue` and `break` may be used to re-iterate the loop and break out of it respectively. The C language also includes the notorious `goto` statement although label names are always identifiers rather than numbers.

One C control construct which you may find more difficult to understand is the `for` loop. Whilst the Pascal version is restricted to stepping through a number range, the C version is much more general. It can be used to step through linked lists, maintain two counters etc. To understand C `for` loops you must keep in mind the general structure of the statement which is:

```
for (initialisation; condition; continuation)
     loop_body;
```

The initialisation part is always performed before the loop begins, typically this is used to set up start values. The loop body will be executed while the condition expression is non-zero. After the loop body, the continuation statement is carried out. This usually updates the loop counters. Any of these sections may be missed out by simply specifying a semicolon.

This may become clear with the aid of a few short examples. The C is first, followed by the Pascal equivalent.

C:
```
for (n = 1; n < 10; n++)
```
Pascal:
```
For n := 1 to 10 do
```

C:
```
for (node = &first; node; node = node.next)
```
Pascal:
```
      node := &first;
While node <> NIL do
      node := node.next;
```

C:
```
for (a = 256; a; a -= 8)
```
Pascal:
```
a:= 256;
while a<0 do begin
      ..........
      a:=a-8;
end;
```

For a more complex example (you will see much worse):

```
for (ptr = &start, i = 0; i != 11; i += *ptr, ptr++)
```

is equivalent to

```
ptr := @start;
i := 0;
While i <> 11 Begin
      i = i + ptr^;
      inc(LongInt(ptr).sizeof(ptr^));
End;
```

Enough of loops. One final control construct is the `case` statement. In C these look like:

```
select (action)
{
      case 1:
            error = SaveFile(buffer);
            break;

      case 2:
            CloseBuffer(buffer);
            break;


      default:
            break;
}
```

An important pitfall which you should watch out for is that omitting the `break` statement at the end of a `case` causes control flow to unconditionally fall through to the next `case`. This is often used in 'cunning' ways by advanced C programmers.

The standard C library

We will not attempt to describe the standard C library although there are a few functions singled out for special attention. The first of these is `printf()`.

`Printf` is to C what `WriteLn` is to Pascal. Most formatted output is performed via this single command. The reason why it is noted here is because `printf` formatting is very different from `WriteLn`. It is data driven from a 'format string'. This includes regular text and control characters such as end of line but also uses the percent character as a special marker.

When `printf()` sees a percent sign, it then picks up the next argument in the list and applies formatting to it according to additional characters following the percent sign. It can be used to print strings and numbers in various ways.

```
printf("Hello\n");                  Hello
printf("Number = %d\n", 10);        Number = 10
printf("In hex: %X...", 42};        In hex: 2A...
printf("Name: %s, Age: %d",         Name: Bob, Age: 67
          &person, age);
```

Some other functions of note are the 'str' family. These include `strcpy` (copy a string, note that the *destination* comes first), `strncpy` (length limited copy), `strcmp` (compare two strings, returns zero if the same), `stricmp` (case insensitive comparison) and `strcat` (add a string to the end of another). There are many more.

This concludes our introduction to reading C programmers; we hope that this section helps the examples in the ROM Kernel manuals and other books seem less frightening!

Appendix A HighSpeed Pascal versus other Pascal Compilers

HighSpeed Pascal does not implement exactly the same language as other compilers; the unique features of the Amiga ensure that. Rather than implement the international standard for Pascal (ISO Pascal) we have followed the de facto standard for Pascal on microcomputers, Borland International's Turbo Pascal for the PC.

This section will cover the main differences between HighSpeed Pascal for the Amiga and these two Pascal standards and also the differences with HighSpeed Pascal on the Atari.

HighSpeed Pascal versus Turbo Pascal for the PC

HighSpeed Pascal is largely compatible with Turbo Pascal 5.0 for the PC. Note that this does not include the object orientated extensions that Borland implemented in version 5.5 of their compiler.

Notice one important thing: HighSpeed Pascal is running on a 68000 micro processor, while Turbo Pascal is running on a 80x86 processor. Both compilers use the CPU's natural way of storing data in memory. The 68000 does not allow reads and writes at odd addresses if the element size is not a byte. Therefore all variables using more space than a byte are allocated on even addresses. Furthermore the 68000 does save words (and others) in a non reversed order in memory, where Turbo Pascal uses the '86 reversed mode.

If you have transferred a typed data file written by Turbo Pascal to your Amiga you must read it,and then swap the byte order using a `N:=Swap(N)` instruction (this could also have been done in Turbo Pascal before writing the data). Long integers must be swapped by using both `SwapWord`, `Swap` and maybe `HiWord` instructions.

The word alignment does make record structures look different when made with either HighSpeed Pascal or Turbo Pascal, even if you use the $A+ option for word alignment. In Turbo Pascal a `string[2]` only uses 3 bytes, while a string in HighSpeed Pascal always uses an even number of bytes, and that makes a `string[2]` use 4 bytes. All strings of length 1 to 3 can then be moved quickly with a single `MOVE` assembly instruction.

HighSpeed Pascal has an extension for omitting the assign statement but including the filename in read/rewrite statements but this is non-standard.

HighSpeed Pascal allows `BlockRead` and `BlockWrite` on all typed files in addition to their use on un-typed files. The parameters given to them are always byte counts.

In Turbo Pascal you can stop a `FOR` loop by assigning the ending value to the control variable. This does not work in HighSpeed Pascal and is explicitly disallowed by ISO Pascal.

HighSpeed Pascal does not support the computed constants of Turbo Pascal.

The use of `NIL` pointers is not checked in Turbo Pascal, but it can wreak havoc on the Amiga - if you have an Amiga with an MMU, our debugger can detect this type of access for you. We also recommend the use of Commodore's Enforcer program on such machines.

The PC uses `CHR(13),CHR(10)` pairs to signify the end of a line rather than the single `CHR(10)` character that is used by the Amiga. This difference only affects programs that explicitly use these characters: if you use `Eoln` and `Readln` to handle line ends you can use exactly the same program on both machines.

Whilst compatibility with Turbo Pascal has been our aim with the DOS, CRT and Graph units, the hardware and operating system differences mean that some things behave differently. For example, the PC's system of wildcards is different to that of the Amiga as is the method for storing packed times. See the appropriate section of this manual for details. The Amiga also tends to use much longer filenames than are common in the PC world so we have lengthened some of the file and pathname handling types.

We have not emulated the use of Borland's fonts in our `Graph` unit since the fonts themselves are copyright of Borland International.

HighSpeed Pascal Amiga vs HighSpeed Pascal Atari

Naturally the implementation of HighSpeed Pascal on Atari computers is the most similar Pascal to HighSpeed Pascal on the Amiga; both use the 68000 processor but they have very different operating systems.

The Atari uses `CHR(13),CHR(10)` pairs to signify the end of a line rather than the single `CHR(10)` character that is used by the Amiga. This difference only affects programs that explicitly use these characters: if you use `Eoln` and `Readln` to handle line ends you can use exactly the same program on both machines.

The DOS unit on the Atari mimics the Turbo Pascal version very well because of the similarities of the operating systems. The differences between the DOS unit on the PC and Amiga apply equally between the Atari and the Amiga.

Conversely, the Graph unit on the Amiga is able to emulate the PC graphics modes much more closely than the Atari version thanks to the flexibility of the Amiga's video hardware.

On the Atari, the use of `NIL` pointers is enforced by the hardware even on 68000 machines; thus an Amiga program that inadvertently reads from a `NIL` pointer won't work on the Atari.

HighSpeed Pascal versus ISO Pascal

HighSpeed Pascal cannot tell you if any non standard features are used.

The `Put` and `Get` system in ISO Pascal is not implemented.

`Procedures` and `functions` do not take procedure and function parameters in HighSpeed Pascal.

The parameter to `New` and `Dispose` cannot specify which variant part to use. You may use `GetMem` and `FreeMem` instead.

`Pack` and `Unpack` are not implemented.

The `@` symbol is an operator in HighSpeed Pascal, and is the same as the `^` in ANS Pascal.

When reading a `char` variable when `eoln` is true, HighSpeed Pascal returns a Newline (code 10) character where ISO Pascal would return a space.

Extensions to ANS Pascal

HighSpeed Pascal cannot tell you if any non standard features are used.

HighSpeed Pascal implements strings with dynamic length, and routines to operate on them.

HighSpeed Pascal implements the types `ShortInt`, `LongInt`, `Single` and `Double`.

HighSpeed Pascal implements the unit method of modular compilation.

Labels can be both identifiers and the normal digit sequence.

Integer constants can be written in hexadecimal format when preceded with a $.

All character values can be written in character and string constants when preceded with a # or a ^. A decimal or hex value can be written after the #.

The sequence of `Label`, `Const`, `Var`, `Type`, `Procedure` and `Function` declarations can be in any order.

There are three new logical operators: `xor`, `shl` and `shr`. The logical operators also work as bitwise operators.

The `@` operator works like the `Addr` function. In ISO Pascal `@` and `^` are the same.

HighSpeed Pascal implements value and variable typecast.

An `ELSE` clause is allowed as the default case in a `CASE` construction. There is no default in ISO Pascal.

A variable formal parameter in a procedure/function can be typeless.

The list of reserved words can be found in the companion *Technical Reference* manual.

Appendix B
Technical Support

HighSpeed Pascal comes with 30 days free technical support, starting from the date of registration; therefore you should send in your registration card quickly. Technical support is available by telephone during our Technical Support Hour, by letter or by fax.

Should you wish to receive extended technical support, please complete the relevant sections on the registration card, indicating whether you would like to take up the *Silver* or the *Gold* service.

In addition to your name, address and postcode (very important for UK customers), we need payment details before we can accept your extended registration. You can pay by credit card (Mastercard, Eurocard, Access, Visa etc.), UK debit card (Switch, Connect etc.), Eurocheque, UK cheque or Postal Order.

You may have already registered another HiSoft product under our *Gold* or *Silver* service; in this case, there is no need to fill out the payment section.

Appendix C
Bibliography

This bibliography contains our suggestions for further reading on the subject of the Amiga's operating system, the Pascal language and programming in general. The views expressed are our own and as with all reference books there is no substitute for looking at the books in a good bookshop before making a decision.

We can supply a number of the books listed, especially the AmigaDOS titles - see the order form enclosed with the HighSpeed Pascal package for details on obtaining these books.

Amiga

The AmigaDOS Manual, 3rd Edition
Bantam Books [1991]

ISBN 0-553-35403-5, Bantam Books, London.

Amiga User Interface Style Guide
Commodore-Amiga, Inc. [1991]

ISBN 0-201-57757-7, Addison-Wesley Publishing Company, Inc.

Amiga ROM Kernel Reference Manual: Includes & Autodocs, 3rd Edition
Commodore-Amiga, Inc. [1991]

ISBN 0-201-56773-3, Addison-Wesley Publishing Company, Inc.

Amiga ROM Kernel Reference Manual: Libraries, 3rd Edition
Commodore-Amiga, Inc. [1991]

ISBN 0-201-56774-1, Addison-Wesley Publishing Company, Inc.

Amiga ROM Kernel Reference Manual: Devices, 3rd Edition
Commodore-Amiga, Inc. [1991]

ISBN 0-201-56775-X, Addison-Wesley Publishing Company, Inc.

Amiga Hardware Reference Manual, 3rd Edition
Commodore-Amiga, Inc. [1991]

ISBN 0-201-56776-8, Addison-Wesley Publishing Company, Inc.

Interchange File Format Specification
Commodore-Amiga, Inc. [1988]

CATS, West Chester, PA 19380, USA..

Pascal

Pascal from BASIC
Peter Brown [1982]

ISBN 0-201-10158-0, Addison-Wesley Publishing Company, Inc

Pascal Precisely, 2nd Edition
Judy Bishop [1989]

ISBN 0-201-41633-6, Addison-Wesley Publishing Company, Inc

Pascal, An introduction to Methodical Programming, 3rd Edition
W. Findlay and D.A. Watt [1985]

ISBN 0-273-02188-5, Pitman Publishing Ltd.

Software Tools in Pascal
Brian W. Kernighan and P.J. Plauger. [1981]

ISBN 0-201-10342-7, Addison-Wesley Publishing Company, Inc

Turbo Pascal 4.0/5.0 An Introduction tro the Art and Science of Programming, 2nd Edition

Walter J. Savitch. [1989]

ISBN 0-8053-0410-X, Addison-Wesley Publishing Company, Inc

Turbo Pascal, Problem Solving and Program Design, 3rd Edition

Elliot B.Kaufman [1991]

ISBN 0-201-52736-7, Addison-Wesley Publishing Company, Inc

Algorithms & Data Structures

Algorithms

Sedgewick, Robert [1988]

ISBN 0-201-06673-4, Addison-Wesley Publishing Company, Reading, MA, USA.

Compilers: Principles, Techniques and Tools

Aho, Alfred V, Ravi Sethi and Jeffrey D. Ullman [1986]

ISBN 0-201-10194-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Data Structures and Algorithms

Aho, Alfred V, John E. Hopcroft and Jeffrey D. Ullman [1983]

ISBN 0-201-00023-7, Addison-Wesley Publishing Company, Reading, MA, USA.

Fundamental Algorithms

Knuth, Donald E. [1973]

ISBN 0-201-03809-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Seminumerical Algorithms
Knuth, Donald E. [1981]

ISBN 0-201-03822-9, Addison-Wesley Publishing Company, Reading, MA, USA.

Sorting and Searching
Knuth, Donald E. [1973]

ISBN 0-201-03803-X, Addison-Wesley Publishing Company, Reading, MA, USA.

⌥P Print Window

Dumps the current window contents onto the printer or to a file. This command can be aborted by pressing Esc. You can choose the printer device and the filename using the *Preferences* command.

⌥S Split Window

Splits a window vertically i.e. makes it taller or shorter depending on its current state; this may hide or uncover another window. You would normally use this to set up the display as you like it and then save the set-up with the *Save Preferences* command. It can be useful at any time, though, if you would like to see more information in a window or you need another window.

This command has no effect on window 1.

⌥T Type

This command works on windows 1, 2 and 4; it changes the type of the display between register (for window 1), disassembly, memory and source (if a source file has been loaded into the window).

⌥W Widen Window

Splits a window horizontally i.e. makes it wider or narrower depending on its current state; this may hide or uncover another window. You would normally use this to set up the display as you like it and then save the set-up with the *Save Preferences* command. It can be useful at any time, though, if you would like to see more information in a window or you need another window.

This command has no effect on window 1.

⌥Z Zoom Window

This zooms the current window to be full size. Other ⌥ commands are still available and normal size can be achieved by pressing Esc or ⌥Z again.

Index

A

amiga units 141
AmigaDOS 146
audio 161

B

backspace key 43
backups
  editor 52
  in the editor 56
  master disk 10
BCPL 147
Bibliography **199**
block **44**
  delete 43
  goto start/end 45
  marking 44
bookmarks, editor 42
bookmarks, goto, editor 49
books, technical 25, 199
breakpoint, debugger (see debugger, breakpoint)

C

C language 173
case dependency, editor 48
CIA resource 169
clipboard 162
  editor 45
commands
  editor 26
compiler
  using from the command line 78
  using from the editor 67
console 163
copy block, editor 45, 46
copy memory, debugger 126
cursor keys
  debugger 107
  in the editor 39
  with keyboard modifiers 39
cursor, use in the editor 39
cut block, editor 45

D

debugger
  abort 119
  address, set 108
  breakpoint 115
    conditional 116
    count 115
    kill 117
    permanent 115
    remove 117
    set 113, 116, 117
    show 117
    simple 115
    stop 115
  command summary 129
  copy memory 126
  current drive 128
  cursor keys 107
  disassemble 127
  disassembly window 105
  edit 108
  exceptions
    description 132
  executing programs 121
  expressions 98
  fill memory 127
  find 122
  hints 130
  history 118
  labels 102
    list 126
  load binary 120
  load program 120
  load source 121
  lock window 109
  memory window 106
  numbers 99
  print window 110
  register

set 114
window 104
registers 102
running program 122
go 122
running programs 121
save binary 120
screen switching 114
search memory 122
settings 124
single step 121
skip 122
split window 110
symbols 102
terminate 119
trace 122
user screen 114
windows 103
commands 108
disassembly 105
edit 108
lock 109
memory 106
print 110
register 104
source code 107
split 110
type 110
zoom 110
zoom window 110
delete block, editor 46
deleting files, editor 52
deletion of text, editor 42
devices 157
audio 161
clipboard 162
console 163
gameport 163
input 164
keyboard 164
narrator 165
parallel 165
printer 160
SCSI 166
serial 166
timer 167
trackdisk 168
disk 169
disk contents 7
DiskFont 148
disks
backup 10

E

editor 25
automatic indent 56
backups 52, 56
block commands 44
clipboard 45
copy block 45, 46
cut block 45
delete block 46
marking a block 44
paste block 45, 46
print block 47
save block 46
bookmarks 42
centre window 61
clipboard 45
commands 26
cursor keys 39
default settings 27
delete commands 44
delete file 52
deleting text 42
file requester 31
find see search
icons 27
inserting text 52
keyboard shortcuts 28
load text 50
loading settings 60
macros 53
making backups 52
marks 42
menus 34
numeric pad 57
printing 60
project 36
replace all 48
replacing 47
requesters and gadgets 27
resident tools 68
save text 51
search
case dependency 48
control characters 49
next 48
special characters 49
tab 49
searching 47
settings 27, 54
saving 58
starting up 27
sub-menus 34
tab size 55
the window 35
undo line 39

using fonts 62
using the compiler 67
using the cursor 39
using workbench icons 51
window layout 38
windows, switching 37
workbench icons 58
Exec 148
executing programs, debugger 121
expansion library 150
expressions, debugger 98

F

file requester 31
files 146
find next, editor 48
find, debugger 122
find, editor see editor, searching
fonts 62

G

gadgets 155
depth 37
in the editor 27
gameport 163
goto bookmark, editor 49
goto line, editor 49
graphics 150

H

hardware unit 172
hints
debugger 130
history
debugger 118
HSPascal 7

I

icons 58, 152
in the editor 51
IDCMP 156
IFF 153
input device 164
inserting text, editor 52
Intuition library 154

K

keyboard 164
keyboard shortcuts
editor 28
keymap unit 173
keys
backspace 43
del 43

L

labels, debugger 102
layers 156
libmaker 137
libraries 144
line
delete 43
undelete 43
line, goto, editor 49
load binary, debugger 120
load program, debugger 120
load source, debugger 121

M

macros 53
manual, how to use 1
mark block, editor 44
marks, editor 42
master disk, backup 10
memory allocation 149
memory requirements 4
menus 155
menus, in the editor 34
message passing 149

N

narrator device 165
numbers, debugger 99

O

operators
  debugger 98

P

Pascal.lib 137
paste block, editor 45, 46
Potgo 171
print block, editor 47
printer 160
printing
  from the editor 60
project 36
  read only 61

R

README File 14
registers, debugger 102
registration card 14
replace all, editor 48
replacing, editor 47
requesters
  confirmation 33
  file 31
  in the editor 27
resources 168
  CIA 169
  disk 169
  file system 170
Running program
  debugger 122
Running programs, debugger 121

S

save binary, debugger 120
save block, editor 46
saving text, editor 52
screen switching, debugger 114
screens 155
SCSI device 166
search 47
  case dependency 48
  next 48
  replace all 48
  replacing 47
search, debugger 122
serial device 166
settings
  debugger (see debugger, preferences)
  editor 54
single step, debugger 121
skip, debugger 122
source code, debugger 107
speech 157, 165
symbols
  debugger 102

T

tab key 41
tab size 55
technical support **197**
terminate, debugger 119
text
  insert, editor 52
  load, editor 50
  save, editor 51
timers 167
trace, debugger 122
trackdisk 168
translator library 157
tutorial, quick 15
typography 4

U

units
  amiga 141
  example of use 20

W

windows 154
debugger (see debugger, windows)
editor, centre 61
layout 38
switching editor windows 37
the editor 35
workbench unit 173

Z

zoom window, debugger 110

Notes